SIGGRAPH 2002 Course 36 Notes

Real-Time Shading

Presenters

Marc Olano
Member of Technical Staff
SGI

John C. Hart
Associate Professor
Department of Computer Science
University of Illinois, Urbana-Champaign

Wolfgang Heidrich
Assistant Professor
Department of Computer Science
The University of British Columbia

Bill Mark
NVIDIA Corp.

Ken Perlin
Associate Professor
Department of Computer Science
NYU

Course Description:

Shading languages, long valued for off-line rendering and production animation, are just becoming possible on interactive graphics hardware. A wide spectrum of applications are poised to use them -- scientific visualization, product design, games, and more. Initial shading hardware provides only low-level, hard-to-use interfaces. This course explains the variety of techniques to build full shading languages on past, current, and future graphics hardware. Participants will see systems in action and learn basic techniques in a series of technology overviews. The course concludes with a panel session allowing free discussion between the speakers and audience.

Prerequisites:

This course assumes working knowledge of a modern real-time graphics API like OpenGL. The participants are also assumed to be familiar with the concepts of procedural shading and shading languages.

Syllabus

I. Background and Building Blocks
   A. Introduction (Olano - 20 min)
      1. What's real-time?
      2. What's procedural shading?
      3. Why do we want real-time procedural shading?
      4. Overview of hardware shading techniques
   B. Noise (Perlin - 40 min)
      1. Uses of band-limited noise
      2. Consistency vs. the proliferation of different noise functions
      3. A noise function standard
      4. Hardware design
   C. Hardware shading effects (Heidrich - 45 min)
      1. BRDFs and reflectance models
      2. Uses of environment maps
      3. Shadows
      4. Bump mapping
      5. Demo of these effects

---

break

II. Shading Language Systems
   A. In the beginning: the pixel stream editor (Perlin - 30 min)
      1. Shading expressions vs. shading languages
      2. Pixel stream editor architecture
   B. PixelFlow shading (Olano - 40 min)
      1. SIMD rendering technology overview
      2. PixelFlow hardware description
      3. Mapping the pfman language to PixelFlow
      4. OpenGL extensions
      5. What we learned (or should have learned)
      6. PixelFlow shading video
   C. Procedural Solid Texturing (Hart - 35 min)
      1. Real-time hardware for antialiased parameterized solid texturing
      2. Real-time procedural solid texturing software using the solid map
      3. Demo

---

break

   D. Shading through multi-pass rendering (Olano - 35 min)
      1. Multi-pass rendering overview
      2. How multi-pass allows general shading
      3. Limitations of current hardware
      4. OpenGL Shader demo

E. Single pass and multiple complex pass shading (Mark - 40 min)
    1. Vertex and fragment shading
    2. A unified system for programmable vertex and fragment shading
    3. Compiling to programmable fragment hardware (register combiners)
    4. Compiling to programmable vertex hardware
    5. Stanford RTSL demo
F. Sampling procedural shaders (Heidrich - 30 min)
    1. Choosing sampling rates and resolutions
    2. View dependent effects
    3. Hardware rendering as texture

---

break

III. Future
    A. Multi-pass RenderMan (Olano - 40 min)
        1. Necessary hardware extensions
        2. Can it be real-time?
        3. Is RenderMan appropriate for real-time shading?
    B. Analysis of shading pipelines (Hart - 35 min)
        1. Grammar for articulating shading pipelines
        2. Application to existing pipelines
        3. Possibilities for future pipelines
IV. Panel-style Q&A (All - 30 min)

Contents

Chapter 1

Introduction

Marc Olano

Introduction

Marc Olano
SGI

Procedural shading is a proven rendering technique in which a short user-written procedure, called a *shader*, determines the shading and color variations across each surface. This gives great flexibility and control over the surface appearance.

The widest use of procedural shading is for production animation, where has been effectively used for years in commercials and feature films. These animations are rendered in software, taking from seconds to hours per frame. The resulting frames are typically replayed at 24-30 frames per second.

One important factor in procedural shading is the use of a shading language. A shading language is a high-level special-purpose language for writing shaders. The shading language provides a simple interface for the user to write new shaders. Pixar's RenderMan shading language [Upstill90] is the most popular, and several off-line renderers use it. A shader written in the RenderMan shading language can be used with any of these renderers.

Meanwhile, polygon-per-second performance has been the major focus for most *interactive* graphics hardware development. Only in the last few years has attention been given to surface shading quality for interactive graphics. Recently, great progress has been made on two fronts toward achieving real-time procedural shading. This course will cover progress on both. First, graphics hardware is capable of performing more of the computations necessary for shading. Second, new languages and *machine abstractions* have been developed that are better adapted for real-time use.

Interactive graphics machines are complex systems with relatively limited lifetimes. Just as the RenderMan shading language insulates the shading writer from the implementation details of the off-line renderer, a real-time shading system presents a simplified view of the interactive graphics hardware. This is done in two ways. First, we create an abstract model of the hardware. This abstract model gives the user a consistent high-level view of the graphics process that can be mapped onto the machine. Second, a special-purpose language allows a high-level description of each procedure. Given current hardware limitations, languages for real-time shading differ quite a bit from the one presented by RenderMan. Through these two, we can achieve *device-independence*, so procedures written for one graphics machine have the potential to work on other machines or other generations of the same machine.

1. Procedural techniques

Procedural techniques have been used in all facets of computer graphics, but most commonly for surface shading. As mentioned above, the job of a surface shading procedure is to choose a color for each pixel on a surface, incorporating any variations in

color of the surface itself and the effects of lights that shine on the surface. A simple example may help clarify this.

We will show a shader that might be used for a brick wall (Figure 1.1). The wall is to be described as a single polygon with *texture coordinates*. These texture coordinates are not going to be used for image texturing: they are just a pair of numbers that parameterize the position on the surface.

The shader requires several additional parameters to describe the size, shape and color of the brick. These are the width and height of the brick, the width of the mortar between bricks, and the colors for the mortar and brick (see Figure 1.1). These parameters are used to fold the texture coordinates into *brick coordinates* for each brick. These are (0,0) at one corner of each brick, and can be used to easily tell whether to use brick or mortar color. A portion of the brick shader is shown in Figure 1.2 (this shader happens to be written in the *pfman* language, detailed in Chapter 3). In this figure, `ss` and `tt` are local variables used to construct the brick coordinates. The simple bricks that result are shown in Figure 1.3a.

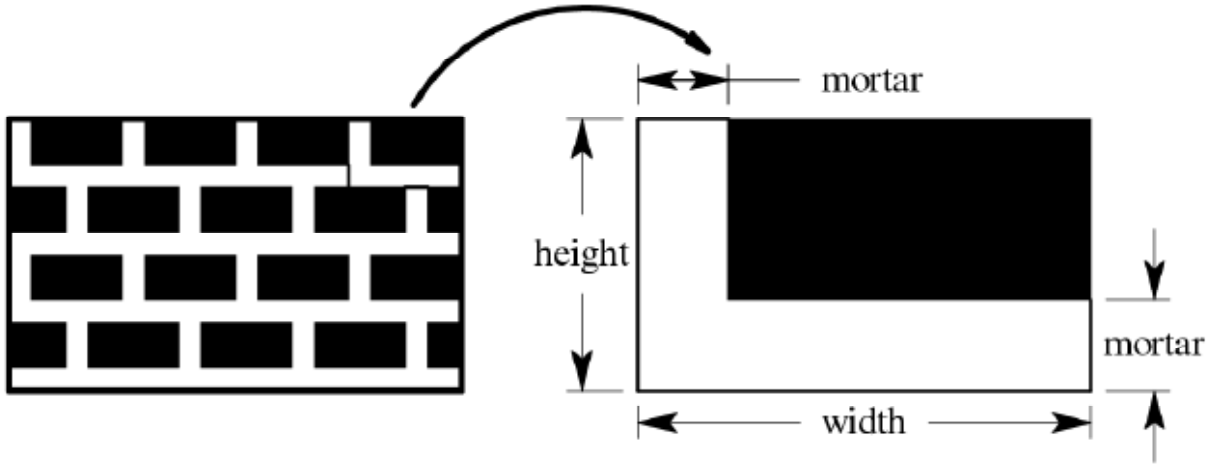


Figure 1.1. Size and shape parameters for brick shader

```
// find row of bricks for this pixel (row is 8-bit integer)
fixed<8,0> row = tt/height;

// offset even rows by half a row
if (row % 2 == 0) ss += width/2;

// wrap texture coordinates to get "brick coordinates"
ss = ss % width;
tt = tt % height;

// pick a color for this pixel, brick or mortar
float surface_color[3] = brick_color;
if (ss < mortar  ||  tt < mortar)
```

```
surface_color = mortar_color;
```

Figure 1.2. Portion of code for a simple brick shader

One of the real advantages of procedural shading is the ease with which shaders can be altered to produce the desired results. Figure 1.3 shows a series of changes from the simple brick shader to a much more realistic brick. Several of these changes demonstrate one of the most common features of procedural shaders: controlled randomness. With controlled use of random elements in the procedure, this same shader can be used for large or small walls without any two bricks looking the same. In contrast, an image texture would have to be re-rendered, re-scanned, or re-painted to handle a larger wall than originally intended.

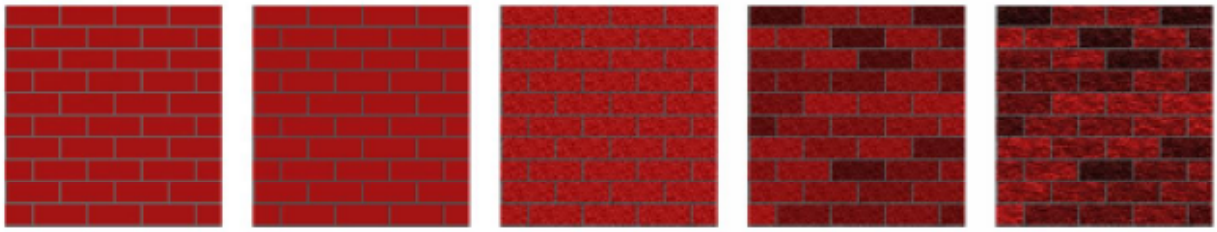


Figure 1.3. Evolution of a brick shader. a) simple version. b) with indented mortar. c) with added graininess. d) with variations in color from brick to brick. e) with color variations within each brick.

Procedural shading can also be used to create shaders that change with time or distance. Figure 1.4a and b are frames from a rippling mirror animated shader. Figure 1.4c shows a yellow brick road where high-frequency elements fade out with distance. Figure 1.4d and e show a wood shader that uses surface position instead of texture coordinates. Figure 1.4d is also lit by a procedural light, simulating light shining through a paned window.

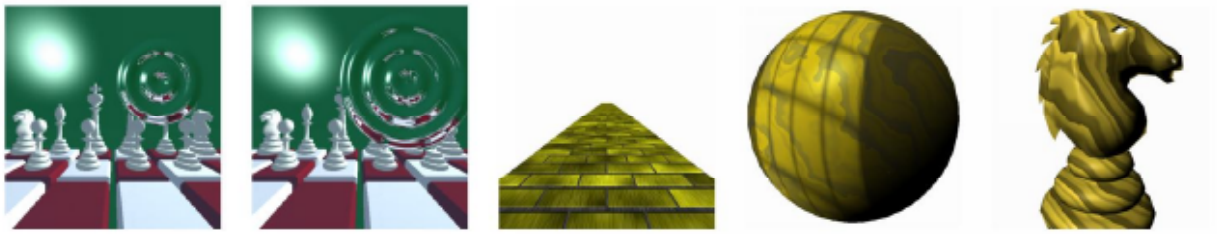


Figure 1.4. Examples of shaders. a+b) two frames of rippling mirror. c) yellow brick road. d+e) wood volume shader.

2. What's to come

These notes are divided into fifteen chapters, following a rough progression from the past of procedural shading, through present-day systems and on to research that may illuminate the future. We provide the following as a rough guide to the connection between chapters in these notes, the course presenters, and what you might expect to find there:

Chapter 1 (Marc Olano): This introduction.

Chapter 2	(Ken Perlin):	Noise, one of the basic building blocks for procedural shading, and how it might be implemented efficiently.
Chapter 3	(Wolfgang Heidrich):	Hardware shading effects, the building blocks for later procedural shading systems.
Chapter 4	(Ken Perlin):	Background on the beginnings of procedural shading and how (even then) it was influenced by hardware concerns.
Chapter 5	(Marc Olano):	The shading capabilities of PixelFlow, the first real-time shading system.
Chapter 6	(John Hart):	Several methods for producing solid textures on hardware.
Chapter 7	(Marc Olano):	Multiple rendering passes using the building blocks from Chapter 3 can be put together to create a full-fledged real-time shading system
Chapter 8	(Bill Mark):	Some of the latest developments in making graphics hardware more flexible and programmable, and a shading language compiler that gives the same high-level interface for both multi-pass shading as introduced in Chapter 7 and shading hardware extensions as introduced in this chapter.
Chapter 9	(Wolfgang Heidrich):	Some issues that make evaluating shading expressions into a texture, one of the most common techniques for real-time shading, harder than it looks.
Chapter 10	(Marc Olano):	How multi-pass rendering techniques could be expanded to support a full-featured shading language like RenderMan.
Chapter 11	(John Hart):	A formal notation for analysis of different real-time shading techniques.
Chapter 12	(All):	A collected bibliography of some of our favorite papers.

Chapter 2

Noise Hardware

Ken Perlin

Noise Hardware

Introduction

Perlin Noise has been a mainstay of computer graphics since 1985 [EBERT98],[FOLEY96],[PERLIN85], being the core procedure that enables procedural shaders to produce natural appearing materials. Now that real-time graphics hardware has reached the point where memory bandwidth is more of a bottleneck than is raw processing power, it is imperative that the lessons learned from procedural shading be adapted properly to real-time hardware platforms.

The original implementation of Noise is fairly simple. First I will show how it is constructed, and various ways it is used in shader programs to get different kinds of procedural textures. There will be pictures and animations. Then I'll talk about several approaches to real-time implementation, so that Noise can be used most effectively in game hardware and other platforms that should support Noise-intensive imagery at many frames per second.

What's Noise?

Noise appears random, but isn't really. If it were really random, then you'd get a different result every time you call it. Instead, it's "pseudo-random" - it gives the *appearance* of randomness.

Noise is a mapping from R^n to R - you input an n-dimensional point with real coordinates, and it returns a real value. Currently the most common uses are for n=1, n=2, and n=3. The first is used for animation, the second for cheap texture hacks, and the third for less-cheap texture hacks. Noise over R^4 is also very useful for time-varying solid textures, as I'll show later.

Noise is *band-limited* - almost all of its energy (when looked at as a signal) is concentrated in a small part of the frequency spectrum. High frequencies (visually small details) and low frequencies (large shapes) contribute very little energy. Its appearance is similar to what you'd get if you took a big block of random values and blurred it (ie: convolved with a gaussian kernel). Although that would be quite expensive to compute.

First I'll outline the *ideal* characteristics of a Noise function, then I'll review how the original Noise function matches these characteristics (sort of). After that I'll outline how one might implement the original Noise function in hardware, in an optimized way. Finally, I'll show a radically different approach I've been taking to implementing Noise, with an eye toward standardizing on a visually better primitive that can also run much faster in hardware.

There are two major issues involved in this adaptation: *(i)* A common procedural shading abstract *machine language*, to enable the capabilities that were first introduced in [Perlin85], and subsequently adapted by the motion picture special effects industry, and *(ii)* a standard, fast, robust, differentiable and extensible Noise function contained in the instruction set of this abstract machine language. This chapter addresses the second of these two issues.

The *ideal* Noise can be separated from the shortcomings of any particular implementation which aims to approximate this ideal. [Perlin85] outlined a number of characteristics for an ideal Noise. Ideally a hardware-implemented standard would conform to this ideal, without suffering from any shortcomings.

The traditional Noise algorithm, while very useful, had some shortcomings that would be of particular consequence in a real-time setting and in a hardware implementation. I'll describe a new method that suffers from none of these shortcomings. Shortcomings which are addressed include:

- *Lack of a single standard reference implementation:* Unlike the situation to date with software versions of Noise, all implementations should ideally produce the same result for the same input, up to the inherent limitation imposed by limited bit depth, on all platforms and implementations.

- *Requiring many multiplies:* The original formulation of Noise required, as a subset of its component calculations, that a gradient be evaluated at each corner of a cube surrounding the input point. Each gradient evaluation requires an inner product, which costs three multiplies, at each of eight cube vertices, for a total of 24 multiplies. A multiply is expensive in its use of hardware, relative to such other operations as bit manipulation and addition. In a hardware implementation, it would be greatly advantageous to redefine Noise so that it does not require this large number of multiplies.

- *Visually significant anisotropy:* The Noise function is ideally a directionally insensitive (isotropic) signal. However, the original implementation, because it consists of adjoining 3D tricubic patches, contains visible directional artifacts which are an unavoidable consequence of its underlying algorithm. This is also the case for approximations that mimic that implementation, such as nVidia's recent Noise patch [NVIDIA00].

  Specifically, when these implementations of Noise are applied to a rotated domain, a casual viewer of the result can easily pick out the orientation of the rotated coordinate grid. Ideally, it should be impossible for a casual viewer to infer the orientation of the rotated coordinate system, when presented with the texture image produced by Noise applied to a rotated domain.

- *Gradient artifacts:* The original Noise function uses a piecewise cubic blending function $3\,t^2 - 2\,t^3$ in each dimension. When the Noise function uses this blending function, then visually noticable discontinuities appear in the derivative of the Noise function along the 3D tricubic patch boundaries of the Noise function's domain, since the derivative of the derivative of this blending function contains a piecewise constant term.

- *Difficulty of computing a derivative:* The original Noise algorithm contains an associated derivative function which is difficult to compute algorithmically, since it consists of a product of a linear function with three cubic splines. In non-real time applications, the Noise derivative has generally been approximated by evaluating differences of Noise at closely spaced sample points along the three coordinate axes. This has required evaluating the Noise function four times. In a hardware implementation such an approach would not only consume valuable gates, but would be impractical for any method that used less than full floating point precision, since the use of difference methods to compute derivatives requires high bit depth. It is desirable for a hardware Noise standard to possess an associated derivative function that can be computable analytically, at a cost of a relatively modest number of gates. This is particularly important when using Noise to compute normal perturbations and other effects that use the derivative of Noise, as opposed to its value.

- *Need for table memory:* The original Noise algorithm relied on a number of table lookups, which are quite reasonable in a software implementation, but which in a hardware implementation are expensive and constitute a cost bottleneck, particularly when multiple instances of the Noise function are required in parallel. Ideally a Noise implementation should not rely on the presence of tables of significant size.

- *Memory-limited extent of the volume tile:* Noise is generally defined within a repeating volumetric tile. In previous implementations, the extent of this tile has been limited by table size. Ideally Noise should be based on a virtual volume which is scalable, inexpensively, to any extent.

- *Expense of generalizing to higher dimensions:* The original implementation of Noise was based on a cubic lattice. Moving to higher dimensions causes computation cost to more than double with each additional dimension, since it requires moving to an n-dimensional hypercube lattice. In hardware, this cost would be measured as a product of gate count and number of successive instruction cycles. For example, the cost of Noise over four dimensions is at least twice the cost of Noise over three dimensions. Quite soon it will be desirable to extend the standard from 3D Noise to 4D Noise (to account for time-varying volume textures), and thereafter to 5D Noise (to specify textured BRDFs). It is important to address this issue now.

- *Lack of separation between signal and reconstruction:* The original Noise presented the pseudo-random gradient field (its "signal") and its tricubic interpolation scheme (its "reconstruction filter") as a single functional object. It would be greatly advantageous for a method to allow these two operations to be cleanly separable, so that other signals which share the same hardware and abstract machine language can also use this reconstruction filter.

In non-real time applications, in which perfection of the final result is far more important than is processing budget, such as is the case in the use of Noise for motion picture special effects, it is possible to "fudge" some these artifacts by applying Noise multiple times. For example, as I mentioned in my previous chapter, the procedural shaders for the scene depicting ocean waves in the recent film "The Perfect Storm" combined about 200 shader procedures, each of which invoked Perlin Noise. In contrast, for real-time applications, where the cost of every evaluation counts, it is crucial that Noise itself be artifact free, that its derivative be directly computable, and that it incur a relatively small computational cost.

The original algorithm:

To make Noise run fast, I originally implemented it as a pseudo-random spline on a regular cubic grid lattice. Now we'll examine this approach in some detail.

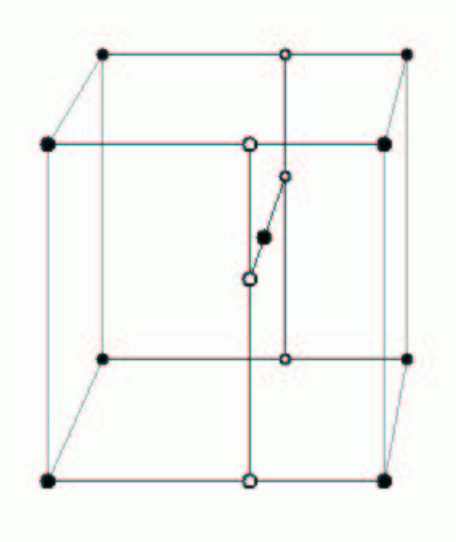
1. Given an input point

2. For each of its neighboring grid points:

    - Pick a "pseudo-random" direction vector
    - Compute linear function (dot product)

3. Linearly combine with a weighted sum, using a cubic ease curve in each dimension, such as $3t^2-2t^3$, as the interpolant.

The 8 neighbors in three dimensions:



In three dimensions, there are eight surrounding grid points. To combine their respective influences we use a trilinear interpolation (linear interpolation in each of three dimensions).

In practice this means that once we've computed the $3t^2-2t^3$ cross-fade function in each of x,y and z, respectively, then we'll need to do seven linear interpolations to get the final result. Each linear interpolation a+t(b-a) requires one multiply.

In the diagram, the six white dots are the results of the first six interpolations: four in x, followed by two in y. Finally, one interpolation in z gives the final result

To compute the pseudo-random gradient, we can first precompute a table of permutations P[n], and a table of gradients G[n]:

$$G = G[(i + P[(j + P[k])_{\bmod n}])_{\bmod n}]$$

A speedier variation would be to let *n* be a power of two. Then, except for bit masking (which is essentially free), the computation reduces to:

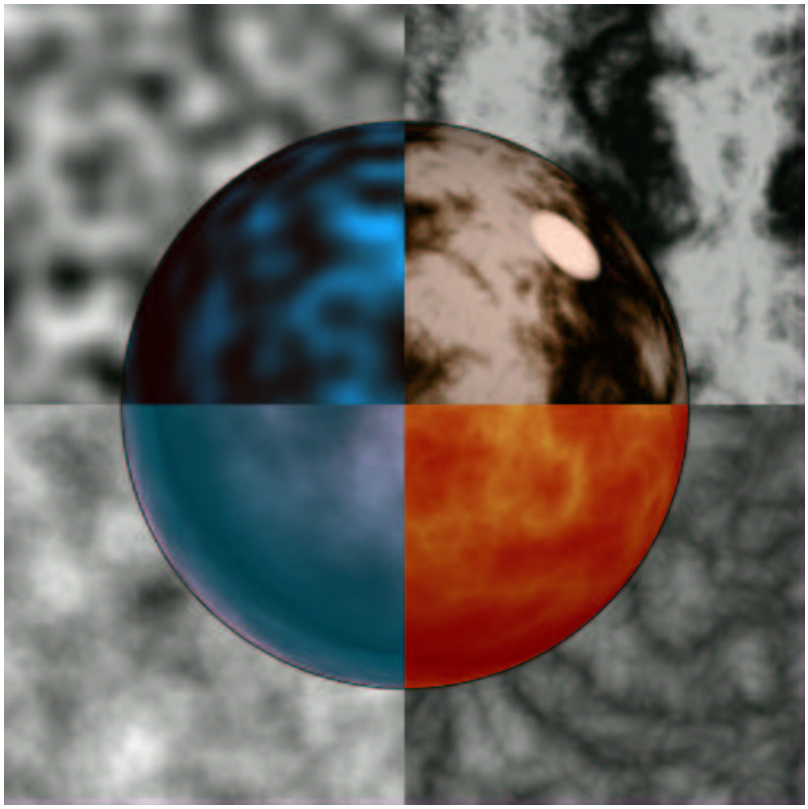
$$G = G[\ \ i + P[\ \ j + P[\ \ k\ \ ]\ \ ]\ \ ]$$

Using in expressions to get texture

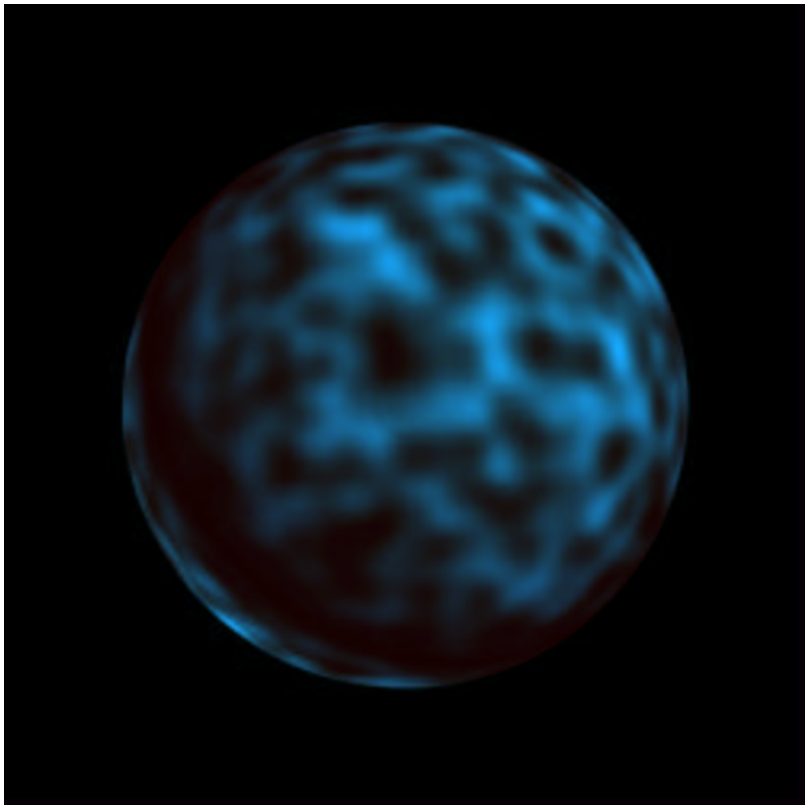
For review, here is a very short set of representative examples to show how Noise can be used to make interesting textures.

- Eg: fractal sums
- 1/f noise: rock, mountains, ...
- 1/f abs(noise): fire, marble, clouds, ...

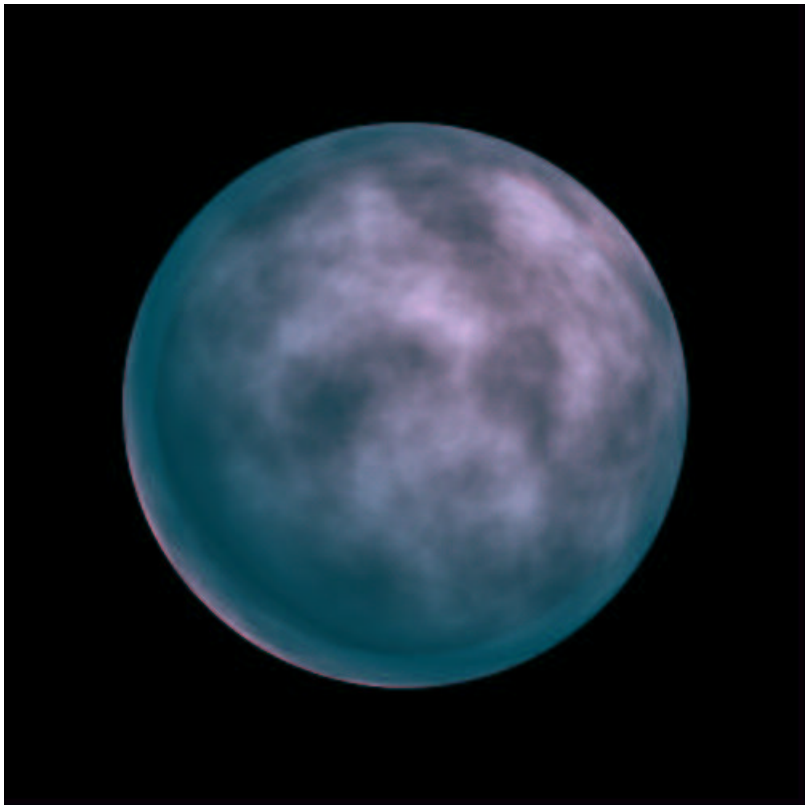
noise sin(x+sum 1/f(|noise|))



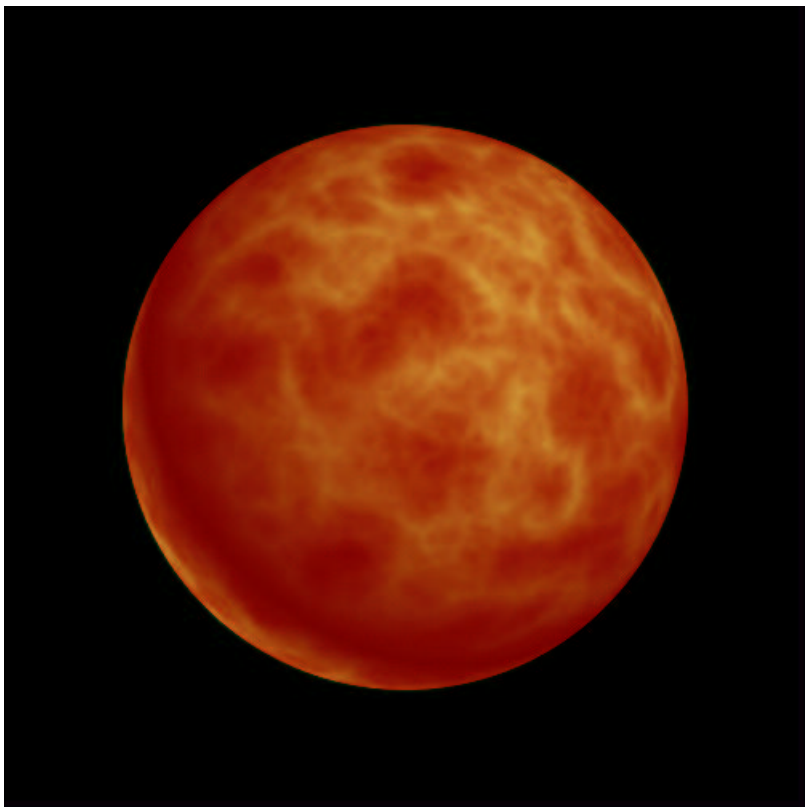
sum 1/f(noise) sum 1/f(|noise|)



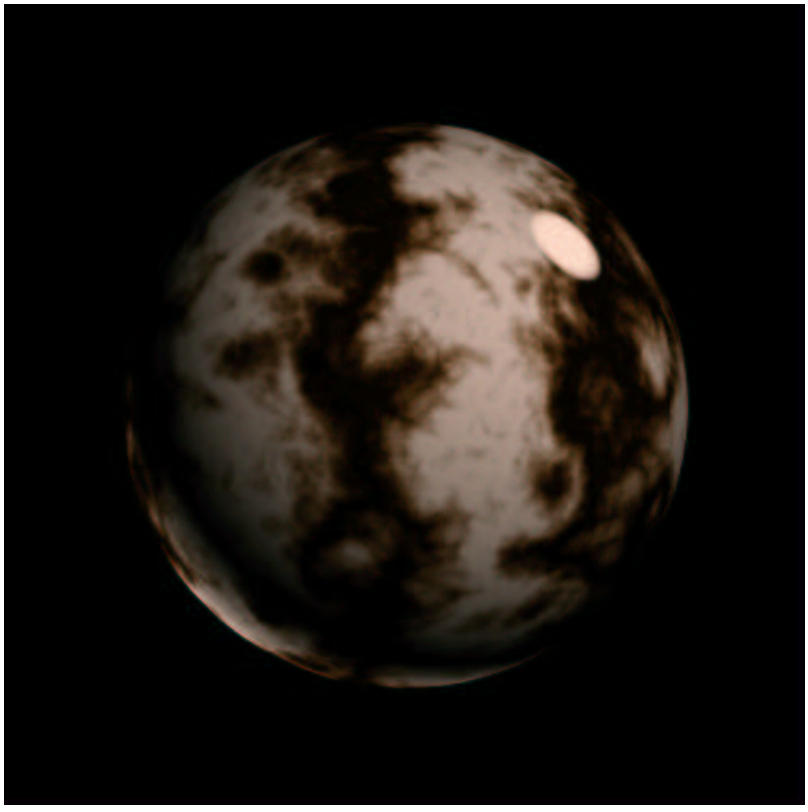
noise



sum 1/f(noise)



sum 1/f(lnoisel)

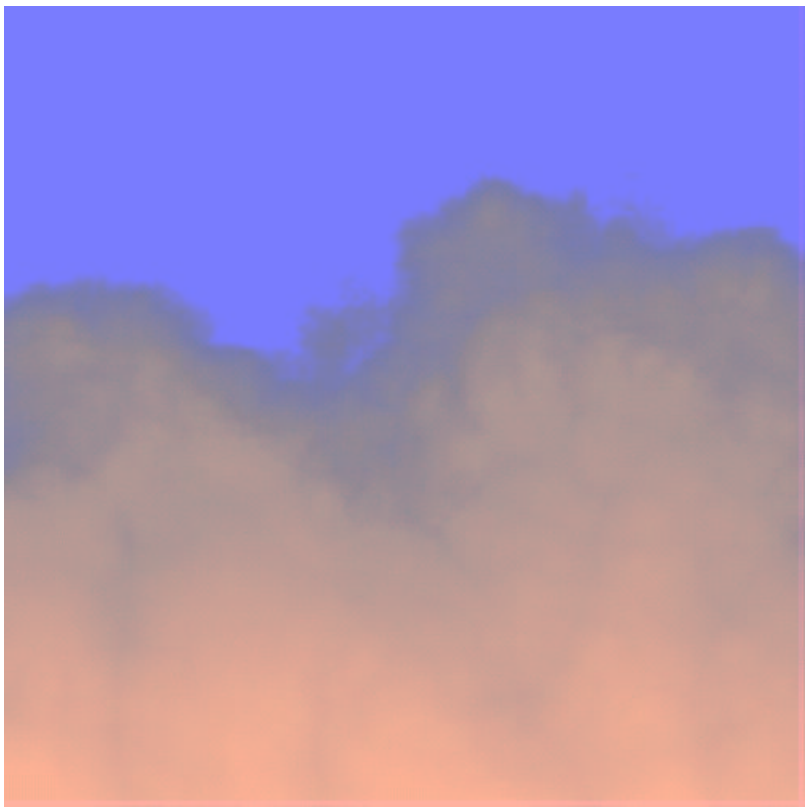


sin(x + sum 1/f(|noise|))

Using 3D noise to animate 2D turbulent flow

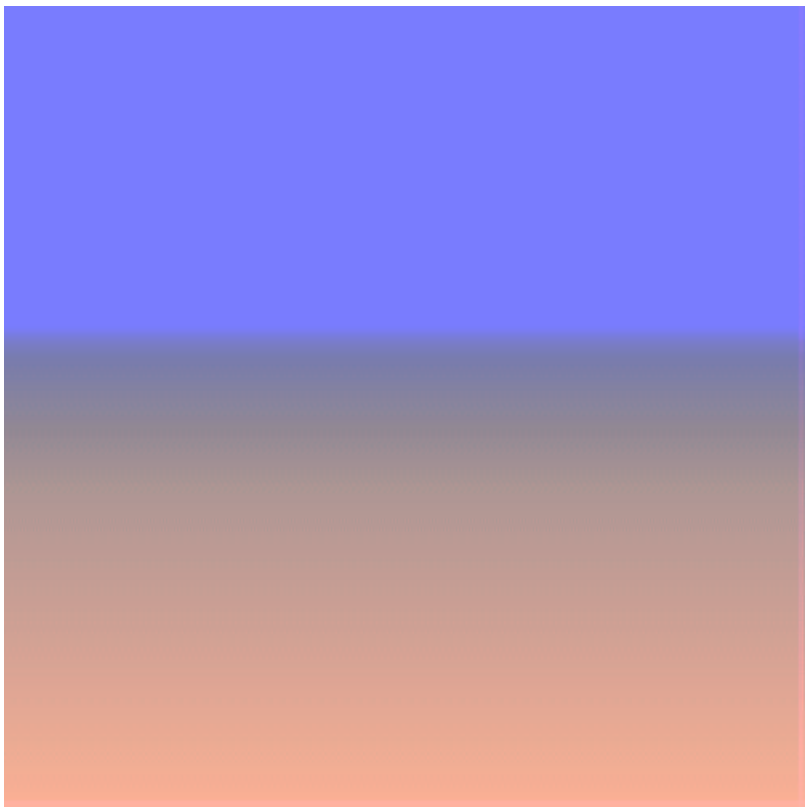
You can also create time-varying animations by using the third dimension of Noise to modulate a texture over time. The following are single frames from pseudo-turbulent animations that were created by dragging, over time, a three dimensional Noise field through the plane z=0. At each animation frame, the intersection of Noise with the (x,y) plane was used to create a flow-perturbation texture. The coherently pseudo-random movement of Noise creates time-coherent pseudo-random flame-like or cloud-like appearance of flow.

Flame: Noise scales in x,y, translates in z



Clouds: Noise translates in x and z

Creating these textures is really quite simple. Below is an image of the gradient field used to create the cloud texture, before perturbation:



Clouds without 1/f(|noise|) perturbation

Optimizing for hardware

Recently I've been looking at the question: what would it take to port the Noise function to a direct hardware implementation? There have been some attempts to speed up Noise by relying on non-traditional instruction sets for various computing chips, first by Intel [INTEL96], to capitalize on their MMX instruction set, then more recently by nVidia [NVIDIA00], to make use of the capabilities of their Vertex processing instruction set (although this only helps for triangle vertices, not for texture samples).

But ideally one would like to implement Noise as a primitive operation, directly in hardware, so that it may be invoked as many times as possible, at the texture sample evaluation level, without becoming a computational bottleneck, and perhaps even for volume filling textures [PERLIN89]. What would this require?

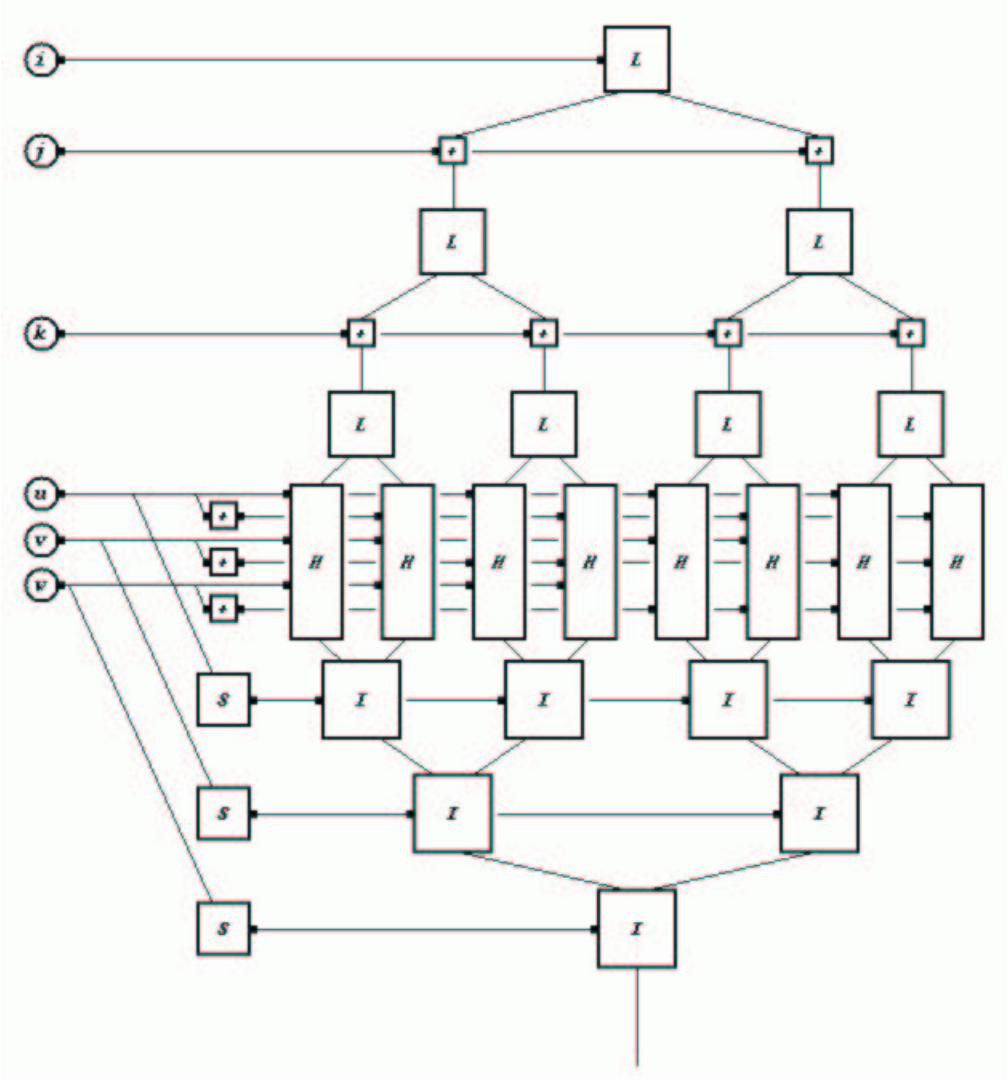
I found that it could be fairly inexpensive, if you worked with a restricted bit depth. For example, if you assume an 8.8 bit input in each of X,Y,Z (ie: each component is in a tiling integer lattice which is 256 units on a side, and there are 256 domain steps within each unit), and an 8 bit output range, then you can implement Noise in under 10K gates, which is a remarkably small number.

The basic approach is to do a pipelined architecture, as in the diagram below. In that diagram:

- *i,j,k* represent the integer (most significant 8 bits) of the input,
- *u,v,w* represent the fractional (least significant 8 bits) of the input,
- *L* represents a computational unit that produces a hash value for one integer component,

- *H* represents a unit that uses this hash value, together with the three fractional components, to produce a contribution from each of the eight corners of the surrounding unit cube,
- *S* represents a unit that computes the cubic interpolant function for each dimension,
- *I* represents a linear interpolator.

The module can be pipelined so that a new Noise evaluation can be calculated at each clock cycle, with a latency of about twenty clock cycles.



A better method:

Ultimately though, the goals I outlined earlier are not satisfied by my original approach to Noise. So I've been developing a new approach. This method of implementing Noise conforms better to the ideal Noise specification of [Perlin85]. While providing the same general "look" as previous versions of Noise, it also:

- provides a single uniform standard result on any platform,
- is visually isotropic, unlike the original algorithm,
- does not require significant table space to compute good pseudo-random gradients,

- can have an arbitrarily large extent for its repeating virtual tile, at very low cost,
- does not require multiplies to evaluate gradient at surrounding grid vertices,
- does not produce visible grid artifacts,
- does not produce visible artifacts in the derivative,
- is cheaper to compute than is the original algorithm,
- allows for a direct analytic computation of derivative at reasonable cost,
- can be generalized to higher dimensions at relatively small computational expense.

I'll describe the method in two parts:

- A pseudo-random signal generator primitive, and
- A recontruction primitive.

This separation enables other signal generators that reside on the same graphics hardware chip to share the same reconstruction hardware and API.

The image below is a side-by-side visual comparison of "traditional" Noise with the method described here. The four quadrants of the image represent, respectively:

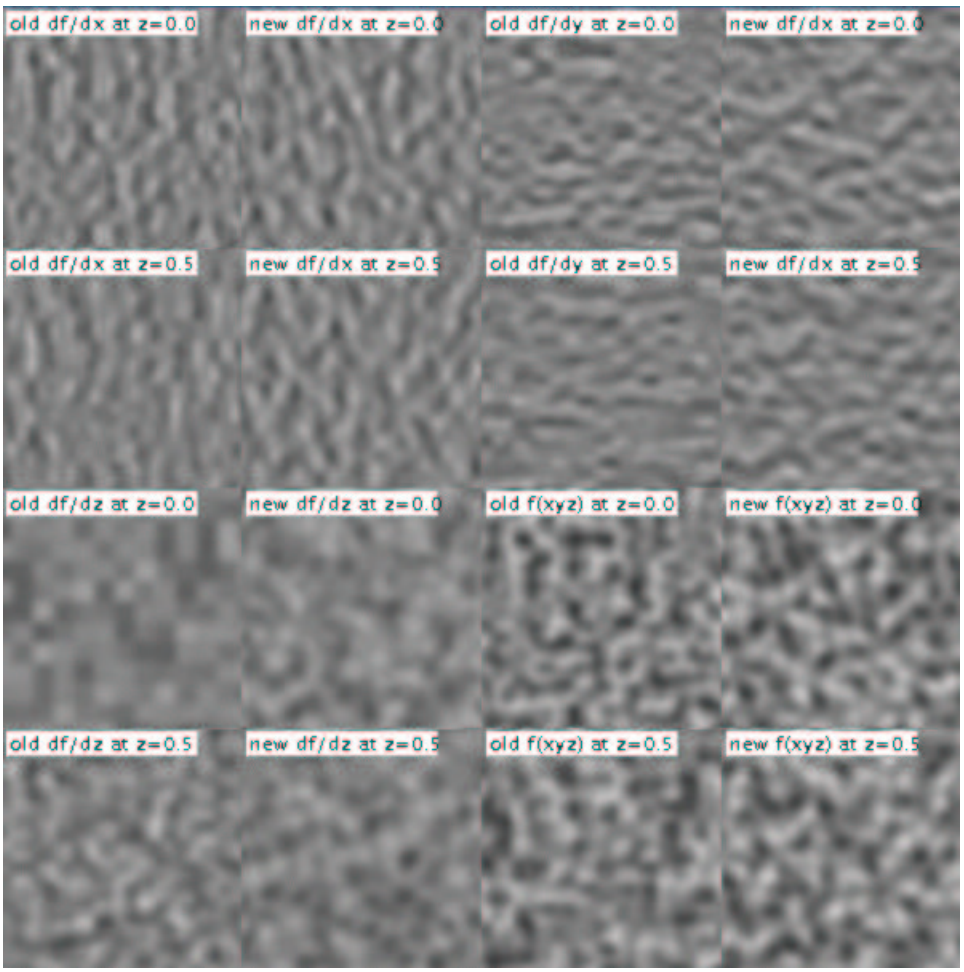
df/dx = d(Noise(xyz))/dx df/dy = d(Noise(xyz))/dy

df/dz = d(Noise(xyz))/dz f(xyz) = Noise(xyz)

Each of these quadrants is divided into four sub-quadrants. These represent, respectively:

old Noise at z = 0 new Noise at z = 0

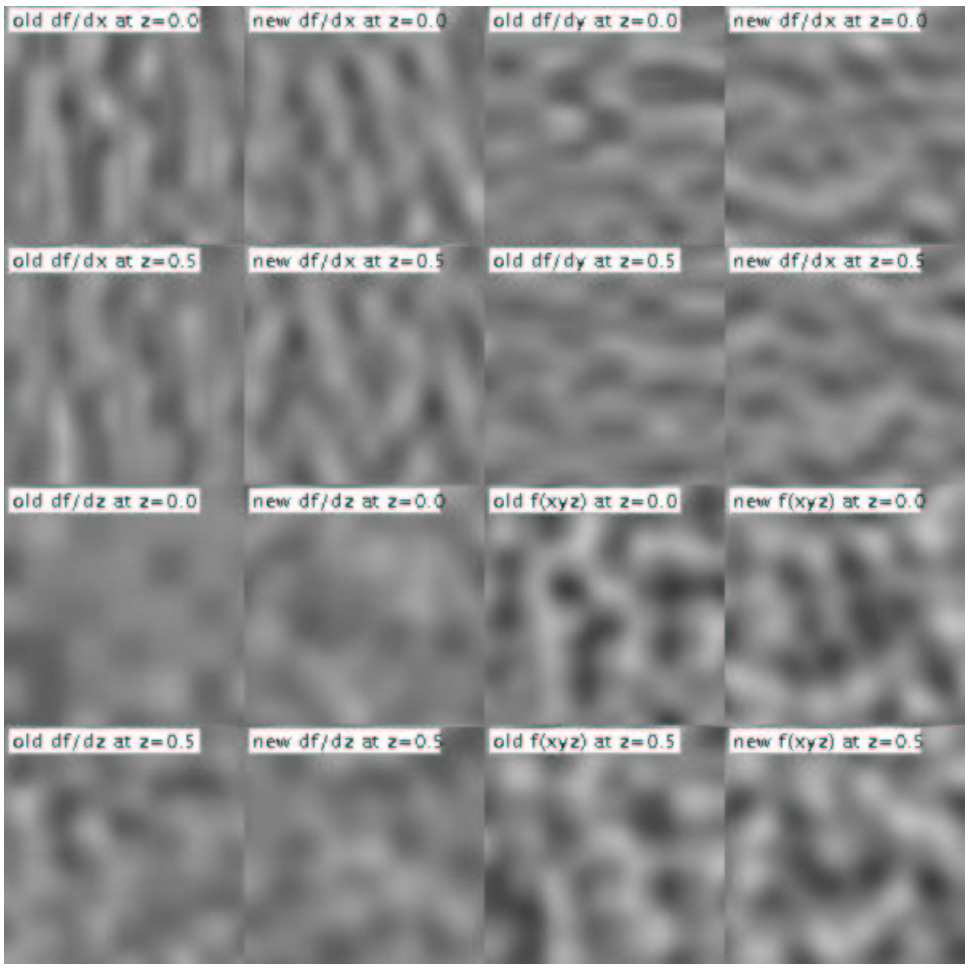
old Noise at z = 0.5 new Noise at z = 0.5



The old and new Noise look roughly the same when evaluated at z=0, the major visual difference being that the new Noise implementation is visually isotropic. Specifically, if the picture is arbitrarily rotated, it is not possible for an observer examining any subportion of the resulting texture to infer, by visual examination, the original orientation of the image produced by new Noise, whereas it is possible for an observer to infer this orientation for the image produced by the old Noise.

Note also the appearance of the derivative with respect to z. In the old Noise this degenerates into an image of fuzzy squares.

Below is the same comparison, this time with the domain magnified by a factor of four. Note the artifacts in df/dx and in df/dy in old Noise, which appear as thin vertical streaks in df/dx and as thin horizontal streaks in df/dy. This is due to the use of the piecewise cubic interpolant function $3\ t^2 - 2\ t^3$, whose derivative is $6\ t - 6\ t^2$, which contains a linear term. The presence of this linear term causes the derivative of the Noise function to be discontinuous at cubic cell boundaries.



Components of the new method:

The new method is a result of combining several different ideas together. When these ideas are used in combination, the result is a far superior Noise function that satisfies all of the requirements outlined above:

1. Rather than using a table lookup scheme to compute the index of a pseudo-random gradient at each surrounding vertex, the new method uses a bit-manipulation scheme that uses only a very small number of hardware gates.

2. Rather than using a cubic interpolation grid, the new method uses a simplicial grid. This confers two advantages during reconstruction:

    1. Only four component evaluations need be done per Noise evaluation (one per contributing vertex), rather than the eight evaluations required in a cubic lattice, and
    2. The axis aligned visual impression of a grid structure is replaced by a far less visually noticable simplicial packing structure.

    Rather than using a tricubic interpolation function, this interpolation scheme uses a spherically symmetric kernel, multiplied by a linear gradient, at each component surrounding vertex. This confers three advantages:

1. The new method contains no directional artifacts due to interpolation function;
2. The new method contains no directional or discontinuity artifacts in gradient;
3. Using the new method, it is practicable to compute the derivative function directly.

Rather than using inner products, with their attendent (and expensive) multiplies, to convert each index into an actual pseudo-random gradient, the new reconstruction method uses a method that produces more visually uniform results, and is easier to integrate into a derivative calculation, while requiring no multiplies at all.

Each of these changes is now described in more detail:

The new pseudo-random generator primitive:

Computing index of pseudorandom gradient

Given an integer lattice point (i,j,k), the new method uses a bit-manipulation algorithm to generate a six bit quantity. This six bit quantity is then used to generate a gradient direction. The six bit quantity is defined as the lower six bits of the sum:

```
b(i,j,k,0) + b(j,k,i,1) + b(k,i,j,2) + b(i,j,k,3) +
b(j,k,i,4) + b(k,i,j,5) + b(i,j,k,6) + b(j,k,i,7)
```

where b() uses its last argument as a bit index into a very small table of bitPatterns.

```
define b(i,j,k,B) :
   patternIndex = 4 * bit_B(i) + 2 * bit_B(j) + bit_B(k)
   return bitPatterns[patternIndex]
```

and where the bit pattern table is defined as:

```
bitPatterns[] = { 0x15,0x38,0x32,0x2c,0x0d,0x13,0x07,0x2a }
```

Using index to derive pseudorandom gradient

The new method converts a six bit pseudo-random index into a visually uniform gradient vector which is easy to integrate into a derivative calculation and which requires no multiplies to compute. The key innovation is to use values of only zero or one for the gradient magnitude.

The specific new technique is as follows: The six bit index is split into *(i)* a lower three bit quantity, which is used to compute a magnitude of either zero or one for each of x,y and z, and *(ii)* an upper three bit quantity, which is used to determine an octant for the resulting gradient (positive or negative sign in each of x,y, and z).

(i) Magnitude computation, based on the three lower bits:

If $bit_1 bit_0 = 0$, then let (p,q,r) = (x,y,z). Otherwise, let (p,q,r) be a rotation of the order of (x,y,z) to (y,z,x) or (z,x,y), as $bit_1 bit_0 = 1$ or 2, respectively, and set either q or r to zero as $bit_2 = 0$ or 1, respectively. The resulting possible rotations are shown in the table below:

$bit_2bit_1bit_0$		$bit_2bit_1bit_0$	
000	p=x q=y r=z	**100**	p=x q=y r=z
001	p=y q=z r=0	**101**	p=y q=0 r=x
010	p=z q=x r=0	**110**	p=z q=0 r=y
011	p=x q=y r=0	**111**	p=x q=0 r=z

(ii) Octant computation, based on the three upper bits:

Once p,q,r have been determined, invert the sign of p if $bit_5 = bit_3$, of q if $bit_5 = bit_4$, and of r if $bit_5 = (bit_4 != bit_3)$, then add together p,q, and r. The resulting possible gradient values are shown in the table below:

$bit_5bit_4bit_3$		$bit_5bit_4bit_3$	
000	-p-q+r	**100**	p+q-r
001	p-q-r	**101**	-p+q+r
010	-p+q-r	**110**	p-q+r
011	p+q+r	**111**	-p-q-r

In this way, a gradient vector is defined using only a small number of bit operations and two additions. In particular, the computation of gradient requires no multiply operations. This contrasts very favourably with previous implementations of Noise, in which three multiply operations were required for each gradient computation (one multiply in each of the three component dimensions).

The new reconstruction primitive:

Simplex grid:

Rather than placing each input point into a cubic grid, based on the integer parts of its (x,y,z) coordinate values, the input point is placed onto a simplicial grid as follows:

1. Skew the input point (x,y,z) to:

```
define skew((x,y,z) -> (x',y',z')) :
   s = (x+y+z)/3
   (x',y',z') = (x+s,y+s,z+s)
```

   This skew transformation linearly scales about the origin, along the x=y=z axis, bringing the point (1,1,1) to the point (2,2,2).

2. Use the integer coordinates in the skewed space to determine a surrounding unit cube whose corner vertex with lowest coordinate values is:

```
(i',j',k') = ( floor(x'),floor(y'),floor(z') )
```

   This corner point can be converted back to the original unskewed coordinate system via:

```
define unskew((i',j',k') -> (i,j,k)) :
```

```
s' = (i'+j'+k')/6
(i,j,k) = (i-s',j-s',k-s')
```

Also consider the original coordinates relative to the unskewed image of the cube corner:

```
(u,v,w) = ( x-i, y-j, z-k )
```

3. Find the simplex containing the point. Relative to (i,j,k), the skewed image of relative point (u,v,w) will lie in one of the six simplices:

```
{ (0,0,0) , (1,0,0) , (1,1,0) , (1,1,1) }
{ (0,0,0) , (1,0,0) , (1,0,1) , (1,1,1) }
{ (0,0,0) , (0,1,0) , (1,1,0) , (1,1,1) }
{ (0,0,0) , (0,1,0) , (0,1,1) , (1,1,1) }
( (0,0,0) , (0,0,1) , (1,0,1) , (1,1,1) }
{ (0,0,0) , (0,0,1) , (0,1,1) , (1,1,1) }
```

Each of these simplices can be defined as an ordered traversal A,B,C from vertex (0,0,0) to vertex (1,1,1) of a unit cube in the skewed space, where { A,B,C } is some permutation of { (1,0,0),(0,1,0),(0,0,1) }. For example, the last simplex above can be defined as an z,y,x traversal, since its first transition A = (0,0,1), its second transition B = (0,1,0), and its third transition C = (0,0,1).

Which simplex contains the input point is determined by the relative magnitudes of u, v and w. For example, if w > v and v > u, then the first transition will be in the z dimension, so A = (0,0,1), and the second transition will be in the y dimension, so B = (0,1,0). In this case, the point lies within the simplex whose traversal order is z,y,x.

The four surrounding vertices of the simplex can now be defined as:

```
{ (i,j,k), (i,j,k)+unskew(A), (i,j,k)+unskew(A+B), (i,j,k)+unskew(A+B+C) }
```

Spherical kernel

If the input point is positioned (u,v,w) from a given simplex vertex, then the contribution from that vertex to the final result will be given by:

```
t = 0.6 - (u^2 + v^2 + w^2)
if t > 0 then 8(t^4) else 0
```

Hardware integration:

The new method can be implemented as a set of pipelined hardware logic gates, in a way that would be very straightforward to design using an FPGA, given the reference implementation below.

Any implementation needs to choose the number of bits of accuracy desired for both input and for output. This choice of bit-depth will vary the number of hardware gates required, but does not in any significant way modify the underlying technique.

In a pipelined implementation, it is very straightforward to maintain a high performance relative to the number of hardware gates used in the implementation, by pipelining the input. This guarantees that the circuitry which implements each different part of the method is always in active use.

The hardware circuitry that implements the new method can make use of an efficiently pipelined parallel implementation, as follows:

1. A supervisory process or driver fills an array with a sequence of (x,y,z) tuples to be evaluated;

2. Noise is invoked by pipelining these input values into a section of logic circuitry that implements it;

3. The resulting sequence, of Noise derivative/value tuples (f_x,f_y,f_z,f), is placed into an output array;

4. The supervisory driver reads out this array of results, and moves on to the next operation within the algorithmic sequence of texture synthesis.

Generalization to higher dimensions:

It is straightforward to generalize this approach to higher dimensions. In n dimensions, a hypercube can be decomposed into n! simplices, where each simplex corresponds to an ordering of the edge traversal of the hypercube from its lower vertex (0,0,...0) to its upper vertex (1,1,...1). For example, when n=4, there are 24 such traversal orderings. To determine which simplex surrounds the input point, one must sort the coordinates in the difference vector (u_1,...u_n) from the lower vertex of the surrounding skewed hypercube to the input point.

For a given n, the "skew factor" f should be set to $f = (n+1)^{1/2}$, so that the point (1,1,...1) is transformed to the point (f,f,...f). In addition, the exact radius and amplitude of the hypersphere-shaped kernel centered at each simplex vertex need to be tuned so as to produce the best visual results for each choice of n.

Previous implementations of Noise, since they were defined on a cubic grid, required a successive doubling of the number of grid points that need to be visited, for each increment in the number of dimensions n. The computational complexity, in terms of vector operations, required to evaluate Noise in n dimensions was therefore $O(2^n)$. Since this is exponential in the number of dimensions, it is not practical beyond a few dimensions.

In contrast, the new implementation , since it is defined on a simplicial grid, requires only an increment in the number of grid points that need be visited, for each increment in the number of dimensions n. The computationational complexity, in terms of vector operations, required to evaluate the new implementation of Noise in n dimensions is therefore O(n). Since this is only polynomial in the number of dimensions, it is practical even for higher dimensions.

To compute the computational complexity in terms of arithmetic operations, both of the above figures need to be multiplied by O(n), since the length of each contributing vector operation, and therefore the computational cost of each vector operation, is n, increasing linearly with the number of dimensions. Therefore the computational complexity of previous Noise implementations in n dimensions is $O(n\ 2^n)$, whereas the computational complexity of the new Noise implementation, in n dimensions is $O(n^2)$.

The important conclusion to be drawn from this analysis is that this new implementation of Noise, in contrast to previous implementations, is practical in even high dimensional spaces, because it is a computation of only *polynomial* complexity, not of *exponential* complexity. For example, the cost of computing Noise in 10 dimensions using previous implementations is approximately $O(10 * 2^{10}) =$

O(10240), whereas the cost using the new implementation is approximately O(10 * 10) = O(100). In this case, a computational advantage factor of 100 is demonstrated. A reference Java implementation of the new Noise algorithm is given below, in Appendix B.

Appendix A

```
/* coherent noise function over 1, 2 or 3 dimensions */
/* (copyright Ken Perlin) */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define B 0x100
#define BM 0xff

#define N 0x1000
#define NP 12   /* 2^N */
#define NM 0xfff

static p[B + B + 2];
static float g3[B + B + 2][3];
static float g2[B + B + 2][2];
static float g1[B + B + 2];
static start = 1;

static void init(void);

#define s_curve(t) ( t * t * (3. - 2. * t) )

#define lerp(t, a, b) ( a + t * (b - a) )

#define setup(i,b0,b1,r0,r1)\
        t = vec[i] + N;\
        b0 = ((int)t) & BM;\
        b1 = (b0+1) & BM;\
        r0 = t - (int)t;\
        r1 = r0 - 1.;

double noise1(double arg)
{
        int bx0, bx1;
        float rx0, rx1, sx, t, u, v, vec[1];

        vec[0] = arg;
        if (start) {
                start = 0;
                init();
        }

        setup(0, bx0,bx1, rx0,rx1);

        sx = s_curve(rx0);

        u = rx0 * g1[ p[ bx0 ] ];
        v = rx1 * g1[ p[ bx1 ] ];

        return lerp(sx, u, v);
}

float noise2(float vec[2])
{
```

```
        int bx0, bx1, by0, by1, b00, b10, b01, b11;
        float rx0, rx1, ry0, ry1, *q, sx, sy, a, b, t, u, v;
        register i, j;

        if (start) {
                start = 0;
                init();
        }

        setup(0, bx0,bx1, rx0,rx1);
        setup(1, by0,by1, ry0,ry1);

        i = p[ bx0 ];
        j = p[ bx1 ];

        b00 = p[ i + by0 ];
        b10 = p[ j + by0 ];
        b01 = p[ i + by1 ];
        b11 = p[ j + by1 ];

        sx = s_curve(rx0);
        sy = s_curve(ry0);

#define at2(rx,ry) ( rx * q[0] + ry * q[1] )

        q = g2[ b00 ] ; u = at2(rx0,ry0);
        q = g2[ b10 ] ; v = at2(rx1,ry0);
        a = lerp(sx, u, v);

        q = g2[ b01 ] ; u = at2(rx0,ry1);
        q = g2[ b11 ] ; v = at2(rx1,ry1);
        b = lerp(sx, u, v);

        return lerp(sy, a, b);
}

float noise3(float vec[3])
{
        int bx0, bx1, by0, by1, bz0, bz1, b00, b10, b01, b11;
        float rx0, rx1, ry0, ry1, rz0, rz1, *q, sy, sz, a, b, c, d, t, u, v;
        register i, j;

        if (start) {
                start = 0;
                init();
        }

        setup(0, bx0,bx1, rx0,rx1);
        setup(1, by0,by1, ry0,ry1);
        setup(2, bz0,bz1, rz0,rz1);

        i = p[ bx0 ];
        j = p[ bx1 ];

        b00 = p[ i + by0 ];
        b10 = p[ j + by0 ];
        b01 = p[ i + by1 ];
        b11 = p[ j + by1 ];

        t  = s_curve(rx0);
        sy = s_curve(ry0);
        sz = s_curve(rz0);

#define at3(rx,ry,rz) ( rx * q[0] + ry * q[1] + rz * q[2] )
```

```
        q = g3[ b00 + bz0 ] ; u = at3(rx0,ry0,rz0);
        q = g3[ b10 + bz0 ] ; v = at3(rx1,ry0,rz0);
        a = lerp(t, u, v);

        q = g3[ b01 + bz0 ] ; u = at3(rx0,ry1,rz0);
        q = g3[ b11 + bz0 ] ; v = at3(rx1,ry1,rz0);
        b = lerp(t, u, v);

        c = lerp(sy, a, b);

        q = g3[ b00 + bz1 ] ; u = at3(rx0,ry0,rz1);
        q = g3[ b10 + bz1 ] ; v = at3(rx1,ry0,rz1);
        a = lerp(t, u, v);

        q = g3[ b01 + bz1 ] ; u = at3(rx0,ry1,rz1);
        q = g3[ b11 + bz1 ] ; v = at3(rx1,ry1,rz1);
        b = lerp(t, u, v);

        d = lerp(sy, a, b);

        return lerp(sz, c, d);
}

static void normalize2(float v[2])
{
        float s;

        s = sqrt(v[0] * v[0] + v[1] * v[1]);
        v[0] = v[0] / s;
        v[1] = v[1] / s;
}

static void normalize3(float v[3])
{
        float s;

        s = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
        v[0] = v[0] / s;
        v[1] = v[1] / s;
        v[2] = v[2] / s;
}

static void init(void)
{
        int i, j, k;

        for (i = 0 ; i < B ; i++) {
                p[i] = i;

                g1[i] = (float)((random() % (B + B)) - B) / B;

                for (j = 0 ; j < 2 ; j++)
                        g2[i][j] = (float)((random() % (B + B)) - B) / B;
                normalize2(g2[i]);

                for (j = 0 ; j < 3 ; j++)
                        g3[i][j] = (float)((random() % (B + B)) - B) / B;
                normalize3(g3[i]);
        }

        while (--i) {
                k = p[i];
                p[i] = p[j = random() % B];
                p[j] = k;
        }
```

```
        for (i = 0 ; i < B + 2 ; i++) {
                p[B + i] = p[i];
                g1[B + i] = g1[i];
                for (j = 0 ; j < 2 ; j++)
                        g2[B + i][j] = g2[i][j];
                for (j = 0 ; j < 3 ; j++)
                        g3[B + i][j] = g3[i][j];
        }
}
```

Appendix B:

A complete implementation of a function returning a value that conforms to the new method is given below as a Java class definition:

```
public final class Noise3 {
   static int i,j,k, A[] = {0,0,0};
   static double u,v,w;
   static double noise(double x, double y, double z) {
      double s = (x+y+z)/3;
      i=(int)Math.floor(x+s); j=(int)Math.floor(y+s); k=(int)Math.floor(z+s);
      s = (i+j+k)/6.; u = x-i+s; v = y-j+s; w = z-k+s;
      A[0]=A[1]=A[2]=0;
      int hi = u>=w ? u>=v ? 0 : 1 : v>=w ? 1 : 2;
      int lo = u< w ? u< v ? 0 : 1 : v< w ? 1 : 2;
      return K(hi) + K(3-hi-lo) + K(lo) + K(0);
   }
   static double K(int a) {
      double s = (A[0]+A[1]+A[2])/6.;
      double x = u-A[0]+s, y = v-A[1]+s, z = w-A[2]+s, t = .6-x*x-y*y-z*z;
      int h = shuffle(i+A[0],j+A[1],k+A[2]);
      A[a]++;
      if (t < 0)
         return 0;
      int b5 = h>>5 & 1, b4 = h>>4 & 1, b3 = h>>3 & 1, b2= h>>2 & 1, b = h & 3;
      double p = b==1?x:b==2?y:z, q = b==1?y:b==2?z:x, r = b==1?z:b==2?x:y;
      p = (b5==b3 ? -p : p); q = (b5==b4 ? -q : q); r = (b5!=(b4^b3) ? -r : r);
      t *= t;
      return 8 * t * t * (p + (b==0 ? q+r : b2==0 ? q : r));
   }
   static int shuffle(int i, int j, int k) {
      return b(i,j,k,0) + b(j,k,i,1) + b(k,i,j,2) + b(i,j,k,3) +
             b(j,k,i,4) + b(k,i,j,5) + b(i,j,k,6) + b(j,k,i,7) ;
   }
   static int b(int i, int j, int k, int B) { return T[b(i,B)<<2 | b(j,B)<<1 | b(k,B
   static int b(int N, int B) { return N>>B & 1; }
   static int T[] = {0x15,0x38,0x32,0x2c,0x0d,0x13,0x07,0x2a};
}
```

References:

[EBERT98] *Texturing and Modeling; A Procedural Approach,* Second Edition; Ebert D. et al, AP Professional; Cambridge 1998c;

[FOLEY96] *Computer Graphics: Principles and Practice,* C version, Foley J., et al, ADDISON-WESLEYD 1996,

[INTEL96], *Using MMX[tm] Instructions for Prtcedural Texture Mapping* Intel Developer Relations Group, Version 1.0, November 18, 1996, http://developer.intel.com/drg/mmx/appnotes/proctex.htm

[NVIDIA00] *TechnicalDemos - Perlin Noise* http://www.nvidia.com/Support/Developer Relations/Technical Demos, Disclosed 11/10/2000.

[PERLIN89] Perlin, K., and Hoffert, E., *Hypertexture,* 1989 Computer Graphics (proceedings of ACM SIGGRAPH Conference); Vol. 23 No. 3.

[PERLIN85] Perlin, K., *An Image Synthesizer,* Computer Graphics; Vol. 19 No. 3.

Chapter 3

Hardware Shading Effects

Wolfgang Heidrich

Real-time Shading: Hardware Shading Effects

Wolfgang Heidrich
The University of British Columbia

Abstract

In this part of the course we will review some examples of shading algorithms that we might want to implement in a real-time or interactive system. This will help us to identify common approaches for real-time shading systems and to acquire information about feature sets required for this kind of system.

The shading algorithms we will look at fall into three categories: realistic materials for local and global illumination, shadow mapping, and finally bump mapping algorithms.

1 Realistic Materials

In this section we describe techniques for a variety of different reflection models to the computation of local illumination in hardware-based rendering. Rather than replacing the standard Phong model by another single, fixed model, we seek a method that allows us to utilize a wide variety of different models so that the most appropriate model can be chosen for each application.

1.1 Arbitrary BRDFs for Local Illumination

We will first consider the case of local illumination, i.e. light that arrives at objects directly from the light sources. The more complicated case of indirect illumination (i.e. light that bounces around in the environment before hitting the object) will be described in Section 1.3.

The fundamental approach for rendering arbitrary materials works as follows. A reflection model in reflection model in computer graphics is typically given in the form of a *bidirectional reflectance distribution function* (BRDF), which describes the amount of light reflected for each pair of incoming (i.e. light) and outgoing (i.e. viewing) direction. This function can either be represented analytically, in which case it is called a reflection model), or it can be represented in a tabular or sampled form as a four-dimensional array (two dimensions each for the incoming and outgoing direction).

The problem with both representations is that they cannot directly be used in hardware rendering: the interesting analytical models are mathematically too complex for hardware implementations, and the tabular form consumes too much memory (a four-dimensional table can easily consume dozens of MB). A different approach has been proposed by Heidrich and Seidel [9]. It turns out that most lighting models in computer graphics can be factored into independent components that only depend on one or two angles. These can then be independently sampled and stored as lower-dimensional tables that consume much less memory. Kautz and McCool [12] described a method for factorizing BRDFs given in tabular form into lower dimensional parts that can be rendered in a similar fashion.

As an example for the treatment of analytical models, consider the one by Torrance and Sparrow [29]:

$$f_r(\vec{l} \rightarrow \vec{v}) = \frac{F \cdot G \cdot D}{\pi \cdot \cos\alpha \cdot \cos\beta}, \qquad (1)$$

where f_r is the BRDF, α is the angle between the surface normal $\vec{n}$ and the vector $\vec{l}$ pointing towards the light source, while β is the angle between $\vec{n}$ and the viewing direction $\vec{v}$. The geometry is depicted in Figure 1.

For a fixed index of refraction, the Fresnel term F in Equation 1 only depends on the angle θ between the light direction $\vec{l}$ and the micro facet normal $\vec{h}$, which is the halfway vector between $\vec{l}$ and $\vec{v}$. Thus, the Fresnel term can be seen as a univariate function $F(\cos\theta)$.

The micro facet distribution function D, which defines the percentage of facets oriented in direction $\vec{h}$, depends on the angle δ between $\vec{h}$ and the surface normal $\vec{n}$, as well as a roughness parameter. This is true for all widely used choices of distribution functions, including a Gaussian distribution of δ or of the surface

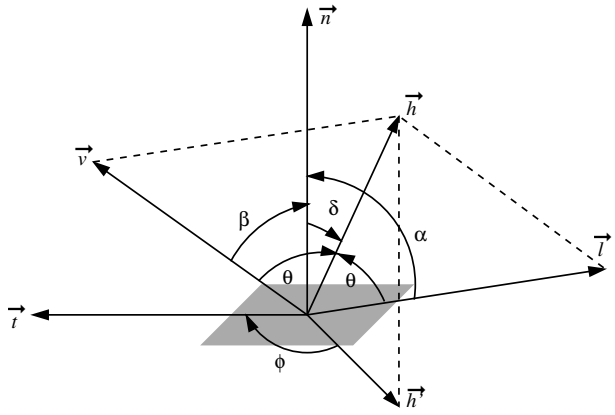


Figure 1: The local geometry of reflection at a rough surface.

height, as well as the distribution by Beckmann [3]. Since the roughness is generally assumed to be constant for a given surface, this is again a univariate function $D(\cos\delta)$.

Finally, when using the geometry term G proposed by Smith [27], which describes the shadowing and masking of light for surfaces with a Gaussian micro facet distribution, this term is a bivariate function $G(\cos\alpha, \cos\beta)$.

The contribution of a single point- or directional light source with intensity I_i to the intensity of the surface is given as $I_o = f_r(\vec{l} \to \vec{v}) \cos\alpha \cdot I_i$. The term $f_r(\mathbf{x}, \vec{l} \to \vec{v}) \cos\alpha$ can be split into two bivariate parts $F(\cos\theta) \cdot D(\cos\delta)$ and $G(\cos\alpha, \cos\beta)/(\pi \cdot \cos\beta)$, which are then stored in two independent 2-dimensional lookup tables.

Regular 2D texture mapping can be used to implement the lookup process. If all vectors are normalized, the texture coordinates are simple dot products between the surface normal, the viewing and light directions, and the micro facet normal. These vectors and their dot products can be computed in software and assigned as texture coordinates to each vertex of the object.

The interpolation of these texture coordinates across a polygon corresponds to a linear interpolation of the vectors without renormalization. Since the reflection model itself is highly nonlinear, this is much better than simple Gouraud shading, but not as good as evaluating the illumination in every pixel (Phong shading). The interpolation of normals without renormalization is commonly known as *fast Phong shading*.

This method for looking up the illumination in two separate 2-dimensional textures requires either a single rendering pass with two simultaneous textures, or two separate rendering passes with one texture each in order to render specular reflections on an object. If two passes are used, their results are multiplied using alpha blending. A third rendering pass with hardware lighting (or a third simultaneous texture) is applied for adding a diffuse term.

If the light and viewing directions are assumed to be constant, that is, if a directional light and an orthographic camera are assumed, the computation of the texture coordinates can even be done in hardware. To this end, light and viewing direction as well as the halfway vector between them are used as row vectors in the texture matrix for the two textures:

$$\begin{bmatrix} 0 & 0 & 0 & \cos\theta \\ h_x & h_y & h_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \cos\delta \\ 0 \\ 1 \end{bmatrix} \quad (2)$$

$$\begin{bmatrix} l_x & l_y & l_z & 0 \\ v_x & v_y & v_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha \\ \cos\beta \\ 0 \\ 1 \end{bmatrix} \quad (3)$$

Figure 2 shows a torus rendered with two different roughness settings using this technique.

We would like to note that the use of textures for representing the lighting model introduces an approximation error: while the term $F \cdot D$ is bounded by the interval $[0, 1]$, the second term $G/(\pi \cdot \cos\beta)$ exhibits a singularity for grazing viewing directions ($\cos\beta \to 0$). Since graphics hardware typically uses a fixed-point representation of textures, the texture values are clamped to the range $[0, 1]$. When these clamped values are used for the illumination process, areas around the grazing angles can be rendered too dark, especially if the surface is very shiny. This artifact can be reduced by dividing the values stored in the texture by a constant which is later multiplied back onto the final result. In practice, however, these artifacts are hardly noticeable.

The same methods can be applied to all kinds of variations of the Torrance-Sparrow model, using different distribution functions and geometry terms, or the approximations proposed in [24]. With varying numbers of terms and rendering passes, it is also possible to come up with similar factorizations for all kinds of other models. For example the Phong, Blinn-Phong

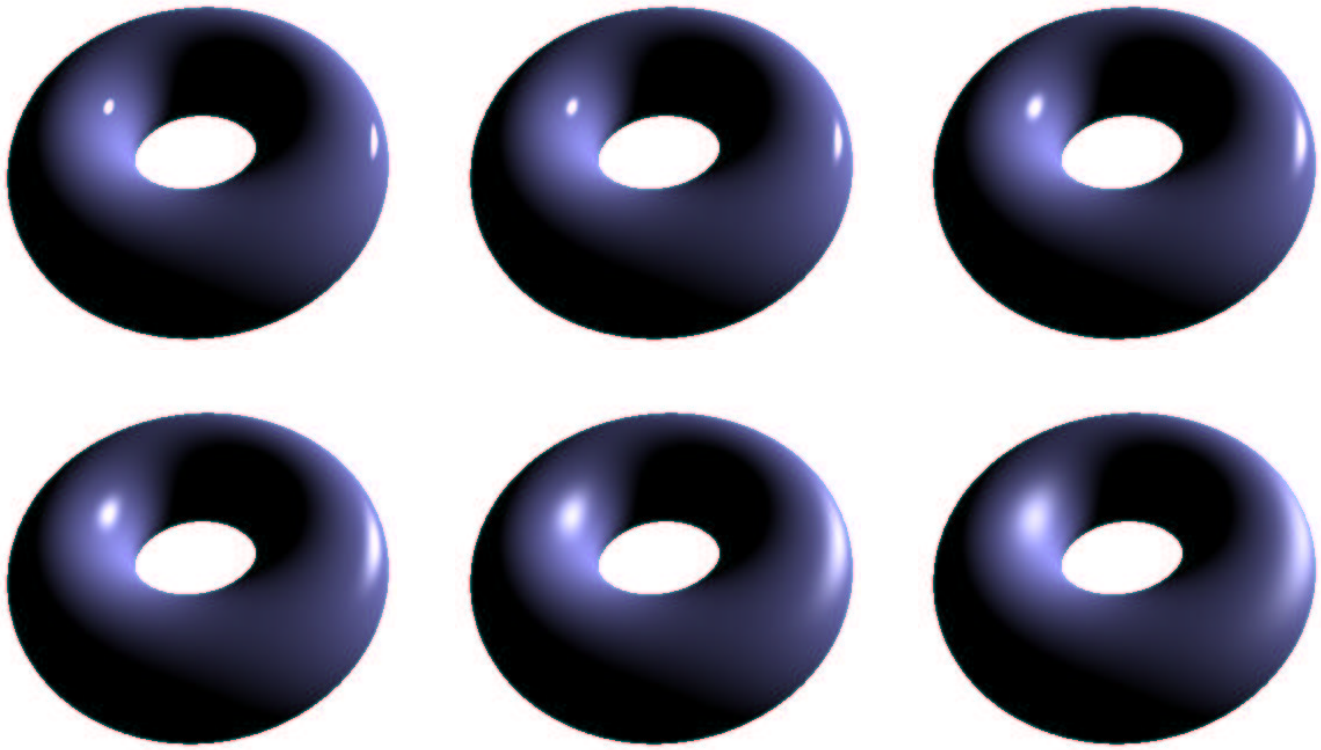


Figure 2: A torus rendered with the proposed hardware multi-pass method using the Torrance-Sparrow reflection model (Gaussian height distribution and geometry term by [27]) and different settings for the surface roughness. For these images, the torus was tessellated into 200 × 200 polygons.

and Cosine Lobe models can all be rendered in a single pass with a single texture, which can even already account for an ambient and a diffuse term in addition to the specular one.

1.1.1 Anisotropy

Although the treatment of anisotropic materials is somewhat harder, similar factorization techniques can be applied here. For anisotropic models, the micro facet distribution function and the geometrical attenuation factor also depend on the angle ϕ between the facet normal and a reference direction in the tangent plane. This reference direction is given in the form of a tangent vector $\vec{t}$.

For example, the elliptical Gaussian model [31] introduces an anisotropic facet distribution function specified as the product of two independent Gaussian functions, one in the direction of $\vec{t}$, and one in the direction of the binormal $\vec{n} \times \vec{t}$. This makes D a bivariate function in the angles δ and ϕ. Consequently, the texture coordinates can be computed in software in much the same way as described above for isotropic materials. This also holds for the other anisotropic models in computer graphics literature.

Since anisotropic models depend on both a normal and a tangent per vertex, the texture coordinates cannot be generated with the help of a texture matrix, even if light and viewing directions are assumed to be constant. This is due to the fact that the anisotropic term can usually not be factored into a term that only depends on the surface normal, and one that only depends on the tangent.

One exception to this rule is the model by Banks [2], which is mentioned here despite the fact that it is an *ad-hoc* model which is not based on physical considerations. Banks defines the reflection off an anisotropic surface as

$$I_o = \cos\alpha \cdot (k_d \langle \vec{n'} | \vec{l} \rangle + k_s \langle \vec{n'} | \vec{h} \rangle^{1/r}) \cdot I_i, \qquad (4)$$

where $\vec{n'}$ is the projection of the light vector $\vec{l}$ into the

plane perpendicular to the tangent vector $\vec{t}$. This vector is then used as a shading normal for a Blinn-Phong lighting model with diffuse and specular coefficients k_d and k_s, and surface roughness r. In [28], it has been pointed out that this Phong term is really only a function of the two angles between the tangent and the light direction, as well as the tangent and the viewing direction. This fact was used for the illumination of lines in [28].

Applied to anisotropic reflection models, this means that this Phong term can be looked up from a 2-dimensional texture, if the tangent $\vec{t}$ is specified as a texture coordinate, and the texture matrix is set up as in Equation 3. The additional term $\cos\alpha$ in Equation 4 is computed by hardware lighting with a directional light source and a purely diffuse material, so that the Banks model can be rendered with one texture and one pass per light source. Figure 3 shows images rendered with this reflection model.

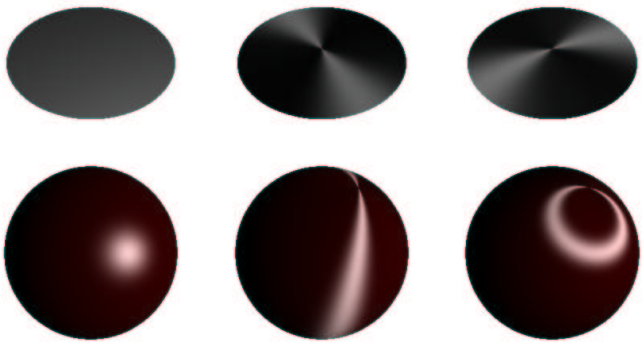


Figure 3: Disk and sphere illuminated with isotropic reflection (left), anisotropic reflection with circular features (center), and radial features (right).

1.1.2 Measured or Simulated Data

As mentioned above, the idea of factorizing BRDFs into low-dimensional parts that can be sampled and stored as textures not only applies to analytical reflection models, but also to BRDFs given in a tabular form. Different numerical methods have been presented for factorizing these tabular BRDFs [12, 18]. The discussion of these is beyond the scope of this course, however.

The advantage of the analytical factorization is that it is very efficient to adjust parameters of the reflection model, so this can be done interactively. The numerical methods take too long for that. On the other hand, the big advantage of the numerical methods is that arbitrary BRDFs resulting from measurements or physical simulations can be used. Figure 4, for example, shows a teapot with a BRDF that looks blue from one side and red from another. This BRDF has been generated using a simulation of micrgeometry [8].

Figure 4: A teapot with a simulated BRDF.

1.2 Global Illumination using Environment Maps

The presented techniques for applying alternative reflection models to local illumination computations can significantly increase the realism of synthetic images. However, true photorealism is only possible if global effects are also considered. Since texture mapping techniques for diffuse illumination are widely known and applied, we concentrate on non-diffuse global illumination, in particular mirror- and glossy reflection.

We describe here an approach based on environment maps, as presented by Heidrich and Seidel [9], because they offer a good compromise between rendering quality and storage requirements. With environment maps, 2-dimensional textures instead of the full 4-dimensional radiance field [19] can be used to store the illumination.

1.3 View-independent Environment Maps

The techniques described in the following assume that environment maps can be reused for different viewing positions in different frames, once they have been generated. It is therefore necessary to choose a representation for environment maps which is valid for arbitrary viewing positions. This includes both cube maps [6]

and parabolic maps [9], both of which are supported on all modern platforms.

1.4 Mirror and Diffuse Terms with Environment Maps

Once an environment map is given in a view-independent parameterization, it can be used to add a mirror reflection term to an object. Using multi-pass rendering and either alpha blending or an accumulation buffer [7], it is possible to add a diffuse global illumination term through the use of a precomputed texture. Two methods exist for the generation of such a texture. One way is, that a global illumination algorithm such as Radiosity is used to compute the diffuse global illumination in every surface point.

The second approach is purely image-based, and was proposed by Greene [6]. The environment map used for the mirror term contains information about the incoming radiance $L_i(\mathbf{x}, \vec{l})$, where $\mathbf{x}$ is the point for which the environment map is valid, and $\vec{l}$ the direction of the incoming light. This information can be used to prefilter the environment map to represent the diffuse reflection of an object for all possible surface normals. Like regular environment maps, this texture is only valid for one point in space, but can be used as an approximation for nearby points.

1.5 Fresnel Term

A regular environment map without prefiltering describes the incoming illumination in a point in space. If this information is directly used as the outgoing illumination, as with regular environment mapping, only metallic surfaces can be modeled. This is because for metallic surfaces (surfaces with a high index of refraction) the Fresnel term is almost one, independent of the angle between light direction and surface normal. Thus, for a perfectly smooth (i.e. mirroring) surface, incoming light is reflected in the mirror direction with a constant reflectance.

For non-metallic materials (materials with a small index of refraction), however, the reflectance strongly depends on the angle of the incoming light. Mirror reflections on these materials should be weighted by the Fresnel term for the angle between the normal and the viewing direction $\vec{v}$.

Similar to the techniques for local illumination presented in Section 1, the Fresnel term $F(\cos\theta)$ for the mirror direction $\vec{r}_v$ can be stored in a texture map. Since here only the Fresnel term is required, a 1-dimensional texture map suffices for this purpose. This Fresnel term is rendered to the framebuffer's alpha channel in a separate rendering pass. The mirror part is then multiplied with this term in a second pass, and a third pass is used to add the diffuse part. This yields an outgoing radiance of $L_o = F \cdot L_m + L_d$, where L_m is the contribution of the mirror term, while L_d is the contribution due to diffuse reflections.

In addition to simply adding the diffuse part to the Fresnel-weighted mirror reflection, we can also use the Fresnel term for blending between diffuse and specular: $L_o = F \cdot L_m + (1 - F)L_d$. This allows us to simulate diffuse surfaces with a transparent coating: the mirror term describes the reflection off the coating. Only light not reflected by the coating hits the underlying surface and is there reflected diffusely.

Figure 5 shows images generated using these two approaches. In the top row, the diffuse term is simply added to the Fresnel-weighted mirror term (the glossy reflection is zero). For a refractive index of 1.5 (left), which approximately corresponds to glass, the object is only specular for grazing viewing angles, while for a high index of refraction (200, right image), which is typical for metals, the whole object is highly specular.

The bottom row of Figure 5 shows two images generated with the second approach. For a low index of refraction, the specular term is again high only for grazing angles, but in contrast to the image above, the diffuse part fades out for these angles. For a high index of refraction, which, as pointed out above, corresponds to metal, the diffuse part is practically zero everywhere, so that the object is a perfect mirror for all directions.

1.6 Precomputed Glossy Reflection and Transmission

We would now like to extend the concept of environment maps to glossy reflections. The idea is similar to the diffuse prefiltering proposed by Greene [6] and the approach by Voorhies and Foran [30] to use environment maps to generate Phong highlights from directional light sources. These two ideas can be combined to precompute an environment map containing the glossy reflection of an object with a Phong material. With this concept, effects similar to the ones presented by Debevec [5] are possible in real time.

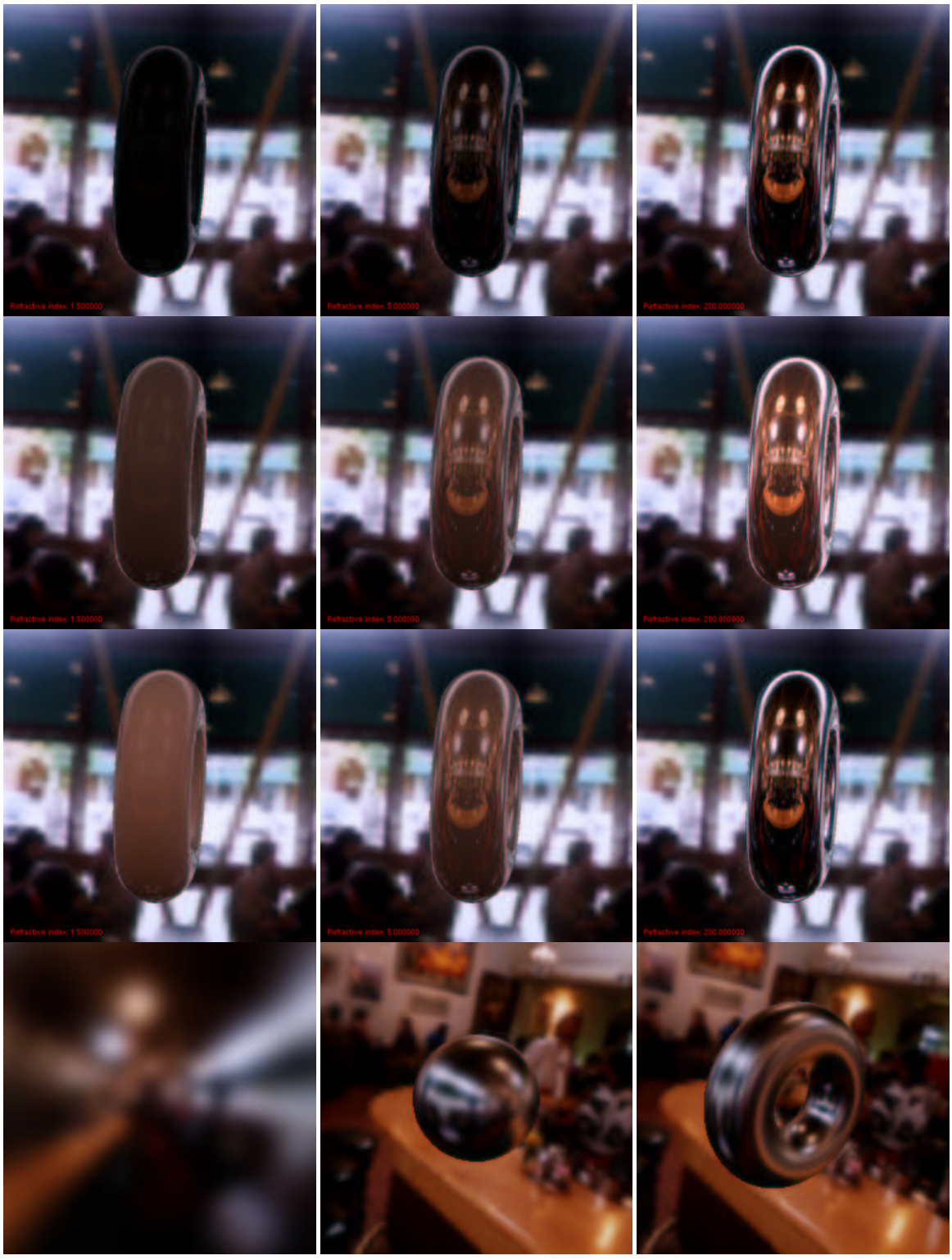


Figure 5: Top row: Fresnel weighted mirror term. Second row: Fresnel weighted mirror term plus diffuse illumination. Third row: Fresnel blending between mirror and diffuse term. The indices of refraction are (from left to right) 1.5, 5, and 200. Bottom row: a prefiltered version of the map with a roughness of 0.01, and application of this map to a reflective sphere and torus.

As shown in [15], the Phong BRDF is given by

$$f_r(\vec{l} \rightarrow \vec{v}) = k_s \cdot \frac{\langle \vec{r}_l | \vec{v} \rangle^{1/r}}{\cos \alpha} = k_s \cdot \frac{\langle \vec{r}_v | \vec{l} \rangle^{1/r}}{\cos \alpha}, \qquad (5)$$

where $\vec{r}_l$, and $\vec{r}_v$ are the reflected light- and viewing directions, respectively.

Thus, the specular global illumination using the Phong model is

$$L_o(\vec{r}_v) = k_s \cdot \int_{\Omega(\vec{n})} \langle \vec{r}_v | \vec{l} \rangle^{1/r} L_i(\vec{l}) \; d\omega(\vec{l}), \qquad (6)$$

which is only a function of the reflection vector $\vec{r}_v$ and the environment map containing the *incoming* radiance $L_i(\vec{l})$. Therefore, it is possible to take a map containing $L_i(\vec{l})$, and generate a filtered map containing the *outgoing* radiance for a glossy Phong material. Since this filtering is relatively expensive, it can on most platforms not be redone for every frame in an interactive application. On special graphics hardware that supports convolution operations, however, it can be performed on the fly, as described by Kautz et al. [13].

The bottom row of Figure 5 shows such a prefiltered map as well as applications of this map for reflection and transmision. If the original environment map is given in a high-dynamic range format, then this prefiltering technique allows for effects similar to the ones described by Debevec [5].

2 Shadow Mapping

After discussing models for local illumination in the previous chapter, we now turn to global effects. In this chapter we deal with algorithms for generating shadows in hardware-based renderings.

Shadows are probably the visually most important global effect. This fact has resulted in a lot of research on how to generate them in hardware-based systems. Thus, interactive shadows are in principle a solved problem. However, current graphics hardware rarely directly supports shadows, and, as a consequence, fewer applications than one might expect actually use the developed methods.

In contrast to the analytic approach shadow volumes, shadow maps [33] are a sampling-based method. First, the scene is rendered from the position of the light source, using a virtual image plane (see Figure 6). The depth image stored in the z-buffer is then used to test whether a point is in shadow or not.

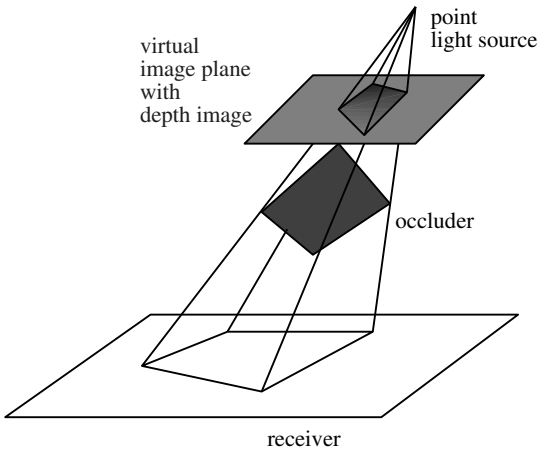


Figure 6: Shadow maps use the z-buffer of an image of the scene rendered from the light source.

To this end, each fragment as seen from the camera needs to be projected onto the depth image of the light source. If the distance of the fragment to the light source is equal to the depth stored for the respective pixel, then the fragment is lit. If the fragment is further away, is is in shadow.

A hardware multi-pass implementation of this principle has been proposed in [25]. The first step is the acquisition of the shadow map by rendering the scene from the light source position. For walkthroughs, this is a preprocessing step, for dynamic scenes it needs to be performed each frame. Then, for each frame, the scene is rendered without the illumination contribution from the light source. In a second rendering pass, the shadow map is specified as a projective texture, and a specific hardware extension is used to map each pixel into the local coordinate space of the light source and perform the depth comparison. Pixels passing this depth test are marked in the stencil buffer. Finally, the illumination contribution of the light source is added to the lit regions by a third rendering pass.

The advantage of the shadow map algorithm is that it is a general method for computing all shadows in the scene, and that it is very fast, since the representation of the shadows is independent of the scene complexity. On the down side, there are artifacts due to the discrete sampling and the quantization of the depth. One benefit of the shadow map algorithm is that the rendering quality scales with the available hardware. The method could be implemented on fairly low end systems, but for high end systems a higher resolution or deeper z-buffer could be chosen, so that the quality in-

creases with the available texture memory. Unfortunately, the necessary hardware extensions to perform the depth comparison on a per-fragment basis are currently only availablehave until recently only been available on two high-end systems, the RealityEngine [1] and the InfiniteReality [20].

2.1 Shadow Maps Using the Alpha Test

Instead of relying on a dedicated shadow map extension, it is also possible to use projective textures and the alpha test. Basically, this method is similar to the method described in [25], but it efficiently takes advantage of automatic texture coordinate generation and the alpha test to generate shadow masks on a per-pixel basis. This method takes one rendering pass more than required with the appropriate hardware extension.

In contrast to traditional shadow maps, which use the contents of a z-buffer for the depth comparison, we use a depth map with a *linear* mapping of the z values in light source coordinates. This allows us to compute the depth values via automatic texture coordinate generation instead of a per-pixel division. Moreover, this choice improves the quality of the depth comparison, because the depth range is sampled uniformly, while a z-buffer represents close points with higher accuracy than far points.

As before, the entire scene is rendered from the light source position in a first pass. Automatic texture coordinate generation is used to set the texture coordinate of each vertex to the depth as seen from the light source, and a 1-dimensional texture is used to define a linear mapping of this depth to alpha values. Since the alpha values are restricted to the range $[0 \ldots 1]$, near and far planes have to be selected, whose depths are then mapped to alpha values 0 and 1, respectively. The result of this is an image in which the red, green, and blue channels have arbitrary values, but the alpha channel stores the depth information of the scene as seen from the light source. This image can later be used as a texture.

For all object points visible from the camera, the shadow map algorithm now requires a comparison of the point's depth with respect to the light source with the corresponding depth value from the shadow map. The first of these two values can be obtained by applying the same 1-dimensional texture that was used for generating the shadow map. The second value is obtained simply by using the shadow map as a projective texture. In order to compare the two values, we can subtract them from each other, and compare the result to zero.

With multi-texturing, this comparison can be implemented in a single rendering pass. Both the 1-dimensional texture and the shadow map are specified as simultaneous textures, and the texture blending function is used to implement the difference. The resulting α value is 0 at each fragment that is lit by the light source, and > 0 for fragments that are shadowed. Then, an alpha test is employed to compare the results to zero. Pixels passing the alpha test are marked in the stencil buffer, so that the lit regions can then be rendered in a final rendering pass.

Without support for multi-texturing, the same algorithm is much more expensive. First, two separate passes are required for applying the texture maps, and alpha blending is used for the difference. Now, the framebuffer contains an α value of 0 at each pixel that is lit by the light source, and > 0 for shadowed pixels. In the next step it is then necessary to set α to 1 for all the shadowed pixels. This will allow us to render the lit geometry, and simply multiply each fragment by $1 - \alpha$ of the corresponding pixel in the framebuffer (the value of $1 - \alpha$ would be 0 for shadowed and 1 for lit regions). In order to do this, we have to copy the framebuffer onto itself, thereby scaling α by 2^n, where n is the number of bits in the α channel. This ensures that $1/2^n$, the smallest value > 0, will be mapped to 1. Due to the automatic clamping to the interval $[0 \ldots 1]$, all larger values will also be mapped to 1, while zero values remain zero. In addition to requiring an expensive framebuffer copy, this algorithm also needs an alpha channel in the framebuffer ("destination alpha"), which might not be available on some systems.

Figure 7 shows an engine block where the shadow regions have been determined using this approach. Since the scene is rendered at least three times for every frame (four times if the light source or any of the objects move), the rendering times for this method strongly depend on the complexity of the visible geometry in every frame, but not at all on the complexity of the geometry casting the shadows. Scenes of moderate complexity can be rendered at high frame rates even on low end systems. The images in Figure 7 are actually the results of texture-based volume rendering using 3D texturing hardware (see [32] for the details of the illumination process).

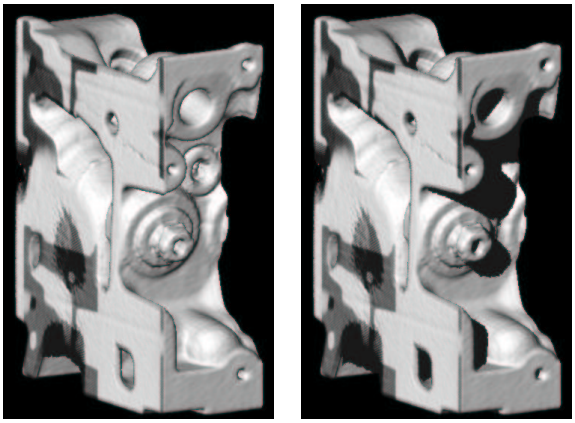


Figure 7: An engine block generated from a volume data set with and without shadows. The shadows have been computed with our algorithm for alpha-coded shadow maps. The Phong reflection model is used for the unshadowed parts.

3 Bump Mapping Algorithms

Bump maps have become a popular approach for adding visual complexity to a scene, without increasing the geometric complexity. They have been used in software rendering systems for quite a while [4], but hardware implementations have only occurred relatively recently, and several different methods are possible, depending on the level of hardware support (e.g. [23, 22, 9, 14]).

The original approach to bump mapping [4] defines surface detail as a height value at every point on a smooth base surface. From this texture-mapped height value, one can compute a per-pixel normal by taking the partial derivatives of the height values. Since this is a fairly expensive operation, most recent hardware implementations [22, 9, 14] precompute the normal for every surface point in an offline process, and store it directly in a texture map.

The bump mapping scheme that has become most popular for interactive applications recently is described in detail in a technical report by Kilgard [14]. First, the light and the viewing vector at every vertex of the geometry is computed and transformed into the local coordinate frame at that vertex ("tangent space", see [22]). In the original version, this is a software step, which can now, however also be done directly in hardware [16]. Then, these local vectors are interpolated across the surface using Gouraud shading and the per-pixel bump map normals are looked up from a texture map. A simple reflection model containing a diffuse and a Phong component can then be implemented as a number of dot products followed by successive squaring (for the Phong exponent). These operations map easily to the register combiner facility present in modern hardware [21].

3.1 Shadows for Bump Maps

The basic approach to bump mapping as outlined above can be extended to approximate the shadows that the bumps cast onto each other. Note that approaches like shadow maps do not work for bump maps because during the rendering phase the geometry is not available; only per-pixel normals are. Shadowing algorithms for bump maps therefore encode the visibility of every surface point for every possible light direction. This is simplified by the fact that bump maps are derived from height fields (i.e. terrains), which allows us to use the notion of a *horizon*. In a terrain, a distant light source located in a certain direction is visible from a given surface point if and only if it is located above the horizon for that surface point. Thus, it is sufficient to encode the horizon for all height field points and directions. This approach is called *horizon mapping*, first presented by Max [17].

The question is, how this horizon information can be represented such that it consumes little memory, and such that the test of whether a given light direction is above or below the horizon for any point in the bump map can be done efficiently in hardware. We describe here a method proposed by Heidrich et al. [8].

We start with a bump map given as a height field, as in the original formulation by Blinn [4]. We then select a number of random directions $D = \{d_i\}$, and shoot rays from all height field points $\mathbf{p}$ into each of the directions d_i. For the shadowing algorithm we will only record a boolean value for each of these rays, namely whether the ray hits another point in the height field, or not. In Section 3.2 we will describe how to use a similar preprocessing step for computing indirect illumination in bump maps.

Now let us consider all the rays shot from a single surface point $\mathbf{p}$. We project all the unit vectors for the sampling directions $\vec{d_i} \in D$ into the tangent plane, i.e. we drop the z coordinate of $\vec{d_i}$ in the local coordinate frame. Then we fit an ellipse containing as many of those 2D points that correspond to unshadowed directions as possible, without containing too many shad-

owed directions. This ellipse is uniquely determined by its (2D) center point $\mathbf{c}$, a direction $(a_x, a_y)^T$ describing the direction of the major axis (the minor axis is then simply $(-a_y, a_x)^T$), and two radii r_1 and r_2, one for the extent along each axis.

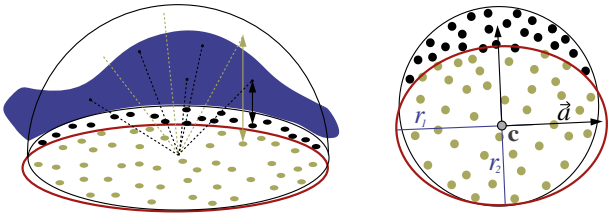


Figure 8: *For the shadow test we precompute 2D ellipses at each point of the height field, by fitting them to the projections of the scattering directions into the tangent plane.*

For the fitting process, we begin with the ellipse represented by the eigenvectors of the covariance matrix of all points corresponding to unshadowed directions. We then optimize the radii with a local optimization method. As an optimization criterion we try to maximize the number of light directions inside the ellipse while at the same time minimizing the number of shadowed directions inside it.

Once we have computed this ellipse for each grid point in the height field, the shadow test is simple. The light direction $\vec{l}$ is also projected into the tangent plane, and it is checked whether the resulting 2D point is inside the ellipse (corresponding to a lit point) or not (corresponding to a shadowed point).

Both the projection and the in-ellipse test can mathematically be expressed very easily. First, the 2D coordinates l_x and l_y have to be transformed into the coordinate system defined by the axes of the ellipse:

$$l'_x := \langle \begin{pmatrix} a_x \\ a_y \end{pmatrix} | \begin{pmatrix} l_x - c_x \\ l_y - c_y \end{pmatrix} \rangle, \tag{7}$$

$$l'_y := \langle \begin{pmatrix} -a_y \\ a_x \end{pmatrix} | \begin{pmatrix} l_x - c_x \\ l_y - c_y \end{pmatrix} \rangle \tag{8}$$

Afterwards, the test

$$1 - \frac{(l'_x)^2}{r_1^2} - \frac{(l'_y)^2}{r_2^2} \geq 0 \tag{9}$$

has to be performed.

To map these computations to graphics hardware, we represent the six degrees of freedom for the ellipses as 2 RGB textures. Then the required operations to implement Equations 7 through 9 are simple dot products as well as additions and multiplications. This is possible using the OpenGL imaging subset [26], available on most contemporary workstations, but also using some vendor specific extensions, such as the *register combiner* extension from NVIDIA [21]. Depending on the exact graphics hardware available, the implementation details will have to vary slightly. These details for different platforms are described in a technical report [11].

Figure 9 shows some results of this shadowing algorithm.

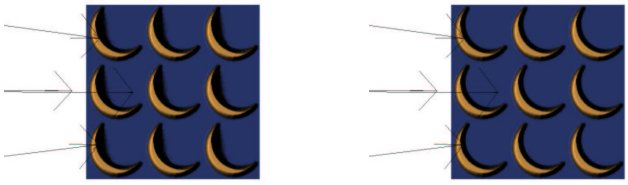


Figure 9: A simple bump map with and without shadows

3.2 Indirect Illumination in Bump Maps

Finally, we would like to discuss a method for computing the indirect light in bump maps [8], i.e. the light that bounces around multiple times in the bumps before hitting the camera.

As in the case of bump map shadows, we start by choosing a set of random directions $d_i \in D$, and shooting rays from al points $\mathbf{p}$ on the height field into all directions d_i. This time, however, we do not only store a boolean value for every ray, but rather the 2D coordinates of the intersection of that ray with the height field (if any). That is, for every direction d_i, we store a 2D map S_i that, for every point $\mathbf{p}$, holds the 2D coordinates of the point $\mathbf{q}$ visible from $\mathbf{p}$ in direction d_i.

Using this precomputed visiblity information, we can then integrate over the light arriving from all directions. For every point $\mathbf{p}$ in the height field, we sum up the indirect illumination arriving from any of the directions d_i, as depicted in Figure 10.

If we assume that both the light and the viewing direction vary slowly across the height field (this corresponds to the assumption that the bumps are relatively small compared to the distance from both the viewer and the light source), then the only strongly varying pa-

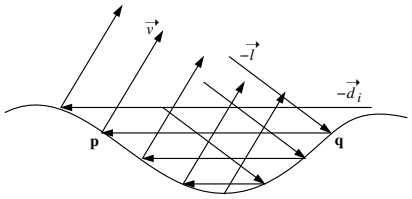


Figure 10: *With the precomputed visibility, the different paths for the illumination in all surface points are composed of pieces with identical directions.*

rameters are the surface normals. More specifically, for the radiance leaving a grid point $\mathbf{p}$ in direction $\vec{v}$, the important varying parameters are the normal $\vec{n}_p$, the point $\mathbf{q} := S_i[\mathbf{p}]$ visible from $\mathbf{p}$ in direction $\vec{d}_i$, and the normal $\vec{n}_q$ in that point.

In particular, the radiance in direction $\vec{v}$ caused by light arriving from direction $\vec{l}$ and scattered once in direction $-\vec{d}_i$ is given by the following formula.

$$L_o(\mathbf{p},\vec{v}) = f_r(\vec{n}_p,\vec{d}_i,\vec{v})\langle\vec{n}_p|\vec{d}_i\rangle \cdot \left(f_r(\vec{n}_q,\vec{l},-\vec{d}_i)\langle\vec{n}_q|\vec{l}\rangle \cdot L_i(\mathbf{q},\vec{l})\right). \quad (10)$$

Usually, the BRDF is written as a 4D function of the incoming and the outgoing direction, both given relative to a local coordinate frame where the local surface normal coincides with the z-axis. In a height field setting, however, the viewing and light directions are given in some global coordinate system that is not aligned with the local coordinate frame, so that it is first necessary to perform a transformation between the two frames. To emphasize this fact, we have denoted the BRDF as a function of the incoming and outgoing direction as well as the surface normal. If we plan to use an anisotropic BRDF on the micro geometry level, we would also have to include a reference tangent vector.

Note that the term in parenthesis is simply the direct illumination of a height field with viewing direction $-\vec{d}_i$, with light arriving from $\vec{l}$. If we precompute this term for all grid points in the height field, we obtain a texture L_d containing the direct illumination for each surface point. This texture can be generated using a bump mapping step where an orthographic camera points down onto the height field, but $-\vec{d}_i$ is used as the viewing direction for shading purposes.

Once we have L_d, the second reflection is just another bump mapping step with $\vec{v}$ as the viewing direction and $\vec{d}_i$ as the light direction. This time, the incoming radiance is not determined by the intensity of the light source, but rather by the content of the L_d texture. For each surface point $\mathbf{p}$ we look up the corresponding visible point $\mathbf{q} = S_i[\mathbf{p}]$. The outgoing radiance at $\mathbf{q}$, which is stored in the texture as $L_d[\mathbf{q}]$, is at the same time the incoming radiance at $\mathbf{p}$.

Thus, we have reduced computing the once-scattered light in each point of the height field to two successive bump mapping operations, where the second one requires an additional indirection to look up the illumination. We can easily extend this technique to longer paths, and also add in the direct term at each scattering point. This is illustrated in the Figure 11.

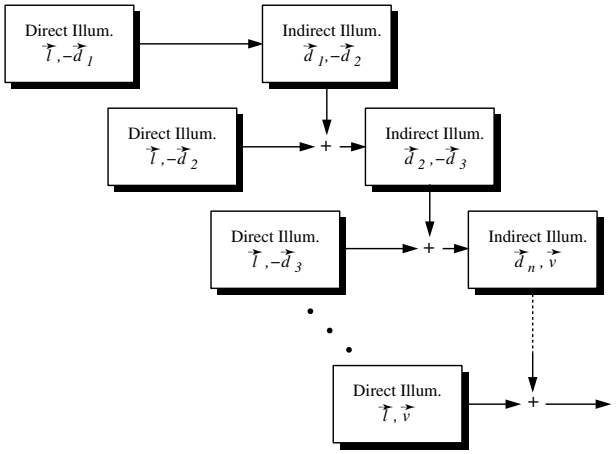


Figure 11: *Extending the dependent test scattering algorithm to multiple scattering. Each box indicates a texture that is generated with regular bump mapping.*

For the total illumination in a height field, we sum up the contributions for several such paths (some 40-100 in most of our scenes). This way, we compute the illumination in the complete height field at once, using two SIMD-style operations on the whole height field texture: bump mapping for direct illumination, using two given directions for incoming and outgoing light, as well as a lookup of the indirect illumination in a texture map using the precomputed visibility data in form of the textures S_i.

3.2.1 Use of Graphics Hardware

In recent graphics hardware, both on the workstation and on the consumer level, several new features have been introduced that we can make use of. In particular, we assume a standard OpenGL-like graphics pipeline [26] with some extensions as described in the following.

Firstly, we assume the hardware has some way of rendering bump maps. This can either be supported

through specific extensions (e.g. [21]), or through the OpenGL imaging subset [26], as described by Heidrich and Seidel [9]. Any kind of bump mapping scheme will be sufficient for our purposes, but the kind of reflection model available in this bump mapping step will determine what reflection model we can use to illuminate our hight field.

Secondly, we will need a way of interpreting the components stored in one texture or image as texture coordinates pointing into another texture. One way of supporting this is the so-called *pixel texture* extension [10, 9], which performs this operation during transfer of images into the frame buffer, and is currently only available on some high-end SGI machines. Alternatively, we can use *dependent texture lookups*, a variant of multi-texturing, that has recently become available on some newer PC graphics boards. With dependent texturing, we can map two or more textures simultaneously onto an object, where the texture coordinates of the second texture are obtained from the components of the first texture. This is exactly the feature we are looking for. In case we have hardware that supports neither of the two, it is quite simple, although not very fast, to implement the pixel texture extension in software: the framebuffer is read out to main memory, and each pixel is replaced by a value looked up from a texture, using the previous contents of the pixel as texture coordinates.

Using these two features, dependent texturing and bump mapping, the implementation of the dependent test method as described above is simple. As depicted in Figure 10, the scattering of light via two points $\mathbf{p}$ and $\mathbf{q}$ in the height field first requires us to compute the direct illumination in $\mathbf{q}$. If we do this for all grid points we obtain a texture L_d containing the reflected light caused by the direct illumination in each point. This texture L_d is generated using the bump mapping mechanism the hardware provides. Typically, the hardware will support only diffuse and Phong reflections, but if it supports more general models, then these can also be used for our scattering implementation.

The second reflection in $\mathbf{p}$ is also a bump mapping step (although with different viewing- and light directions), but this time the direct illumination from the light source has to be replaced by a per-pixel radiance value corresponding to the reflected radiance of the point $\mathbf{q}$ visible from $\mathbf{p}$ in the scattering direction. We achieve this by bump mapping the surface with a light intensity of 1, and by afterwards applying a pixel-wise multiplication of the value looked up from L_d with the help of dependent texturing. Figure 12 shows how to conceptually set up a multi-texturing system with dependent textures to achieve this result.

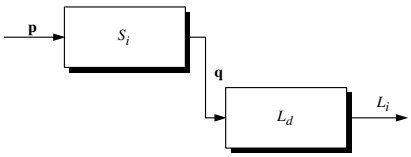


Figure 12: *For computing the indirect light with the help of graphics hardware, we conceptually require a multi-texturing system with dependent texture lookups. This figure illustrates how this system has to be set up. Boxes indicate one of the two textures, while incoming arrows signal texture coordinates and outgoing ones mean the resulting color values.*

The first texture is the S_i that corresponds to the scattering direction d_i. For each point $\mathbf{p}$ it yields $\mathbf{q}$, the point visible from $\mathbf{p}$ in direction d_i. The second texture L_d contains the reflected direct light in each point, which acts as an incoming radiance at $\mathbf{p}$. Figure 13 shows some results of the method.

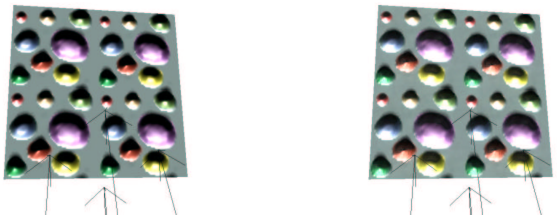


Figure 13: A bump map with and without indirect illumination

By using this hardware approach, we treat the graphics board as a SIMD-like machine which performs the desired operations, and computes one light path for each of the grid points at once. This use of hardware dramatically increases the performance over the software version to an almost interactive rate.

4 Conclusion

In this part, we have reviewed some of the more complex shading algorithms that utilize graphics hardware. While the individual methods are certainly quite different, there are some features that occur in all examples:

- The most expensive operations (i.e. visibility computations, filtering of environment maps etc.) are not performed on the fly, but are done in a precomputing step.
- The results of the precomputation are represented in a sampled (tabular) form that allows us to use texture mapping to apply the information in the actual shaders.
- The shaders themselves are often relatively simple due to the amount of precomputation. They mostly have the job of combining the precomputed textures in various flexible ways.
- The textures need to be parameterized in such a way that the texture coordinates are easy and efficient to generate, ideally directly in hardware.

References

[1] Kurt Akeley. RealityEngine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 109–116, August 1993.

[2] David C. Banks. Illumination in diverse codimensions. In *Computer Graphics (Proceedings of SIGGRAPH '94)*, pages 327–334, July 1994.

[3] Petr Beckmann and Andre Spizzichino. *The Scattering of Electromagnetic Waves from Rough Surfaces*. McMillan, 1963.

[4] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 286–292, August 1978.

[5] Paul E. Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 189–198, July 1998.

[6] Ned Greene. Applications of world projections. In *Proceedings of Graphics Interface '86*, pages 108–114, May 1986.

[7] Paul E. Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, pages 309–318, August 1990.

[8] W. Heidrich, K. Daubert, J. Kautz, and H.-P. Seidel. Illuminating Micro Geometry Based on Precomputed Visibility. In *Computer Graphics (SIGGRAPH '00 Proceedings)*, pages 455–464, July 2000.

[9] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, August 1999.

[10] Silicon Graphics Inc. *Pixel Texture Extension*, December 1996. Specification document, available from http://www.opengl.org.

[11] Jan Kautz, Wolfgang Heidrich, and Katja Daubert. Bump map shadows for OpenGL rendering. Technical Report MPI-I-2000-4-001, Max-Planck-Institut für Informatik, 2000.

[12] Jan Kautz and Michael D. McCool. Interactive rendering with arbitrary BRDFs using separable approximations. In *Rendering Techniques '99 (Proc. of Eurographics Workshop on Rendering)*, pages 247 – 260, June 1999.

[13] Jan Kautz, Pere-Pau Vázquez, Wolfgang Heidrich, and Hans-Peter Seidel. Unified approach to prefiltered environment maps. In *Rendering Techniques '00*.

[14] Mark Kilgard. A practical and robust bump mapping technique. Technical report, NVIDIA, 2000. available from http://www.nvidia.com.

[15] Robert R. Lewis. Making shaders more physically plausible. In *Fourth Eurographics Workshop on Rendering*, pages 47–62, June 1993.

[16] Erik Lindholm, Mark Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Computer Graphics (SIGGRAPH '01 Proceedings)*, August 2001.

[17] Nelson L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4(2):109–117, July 1988.

[18] Anis Ahmad Michael D. McCool, Jason Ang. Homomorphic factorization of BRDFs for high-performance rendering. In *Computer Graphics (SIGGRAPH '01 Proceedings)*, 2001.

[19] Gavin Miller, Steven Rubin, and Dulce Ponceleon. Lazy decompression of surface light fields for precomputed global illumination. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)*, pages 281–292, March 1998.

[20] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A real-time graphics system. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 293–302, August 1997.

[21] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, October 1999. Available from http://www.nvidia.com.

[22] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 303–306, August 1997.

[23] Andreas Schilling, Günter Knittel, and Wolfgang Straßer. Texram: A smart memory for texturing. *IEEE Computer Graphics and Applications*, 16(3):32–41, May 1996.

[24] Christophe Schlick. A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering*, pages 73–83, June 1993.

[25] Marc Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadow and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

[26] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2)*, 1998.

[27] Bruce G. Smith. Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation*, 15(5):668–671, September 1967.

[28] Detlev Stalling, Malte Zöckler, and Hans-Christian Hege. Fast display of illuminated field lines. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):118–128, 1997.

[29] Kenneth E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America*, 57(9):1105–1114, September 1967.

[30] D. Voorhies and J. Foran. Reflection vector shading hardware. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 163–166, July 1994.

[31] Gregory J. Ward. Measuring and modeling anisotropic reflection. *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 265–273, July 1992.

[32] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 169–178, July 1998.

[33] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, August 1978.

Chapter 4

In the beginning: The Pixel Stream Editor

Ken Perlin

4. In the beginning: the pixel stream editor

Procedural texture

Combining controlled noise into various mathematical expressions produces *procedural texture* [EBERT98],[FOLEY96],[PERLIN85].

Unlike traditional texture mapping, procedural texture doesn't require a source texture image. As a result, the bandwidth requirements for transmitting or storing procedural textures are essentially zero.

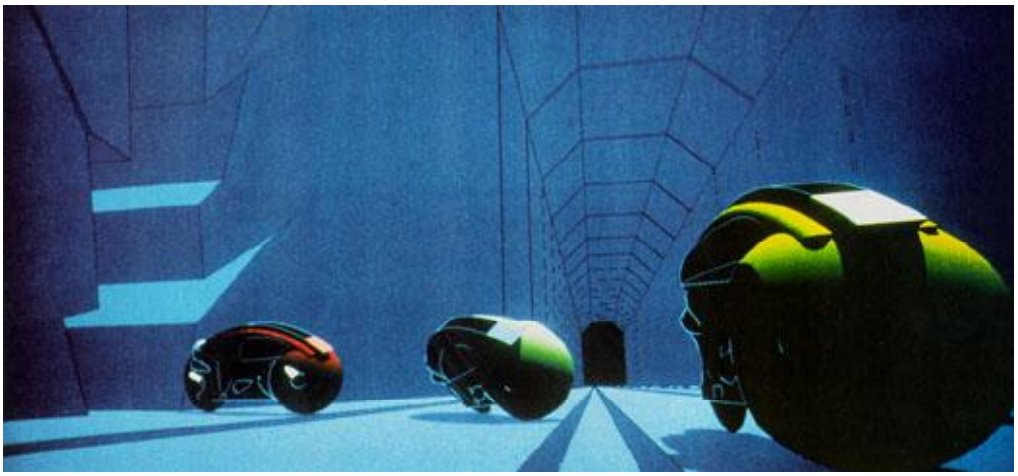
Also, procedural texture can be applied directly onto a three dimensional object. This avoids the "mapping problem" of traditional texture mapping. Instead of trying to figure out how to wrap a two dimensional texture image around a complex object, you can just dip the object into a soup of procedural texture material, defined as a function over a volumetric domain. Essentially, the virtual object is carved out of a virtual solid material defined by the procedural texture. For this reason, procedural texture is sometimes called *solid texture*.

TRON

I first started to think seriously about procedural textures when I was working on TRON at MAGI in Elmsford, NY, in 1981. TRON was the first movie with a large amount of solid shaded computer graphics. This made it revolutionary. On the other hand, the look designed for it by its creator Steven Lisberger was based around the known limitations of the technology.

Lisberger had gotten the idea for TRON after seeing the MAGI demo reel in 1978. He then approached the Walt Disney Company with his concept. Disney's Feature Film division was then under the visionary guidance of Tom Wilhite, who arranged for contributions from the various computer graphics companies of the day, including III, MAGI, and Digital Effects.

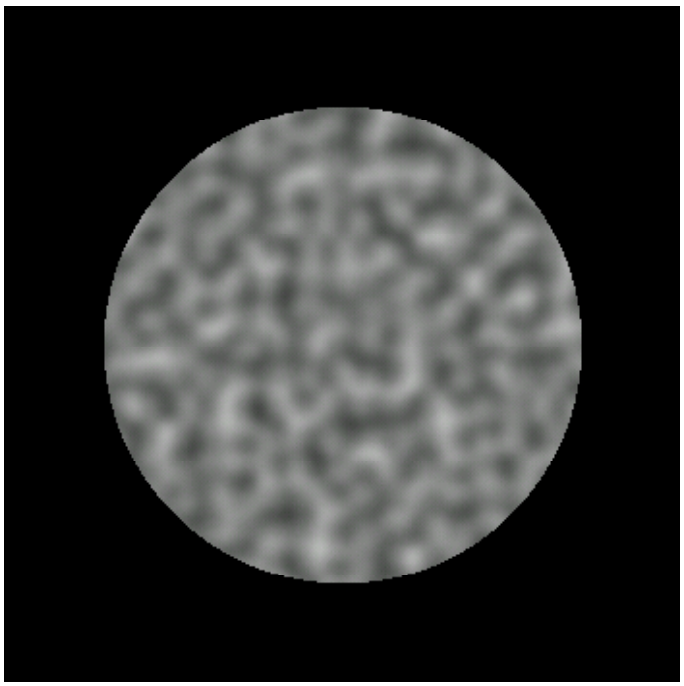
Working on TRON was a blast, but on some level I was frustrated by the fact that everything looked machine-like (a typical scene is shown below). In fact, that machine-like aesthetic became the "look" associated with MAGI in the wake of TRON. So I got to work trying to help us break out of the "machine-look ghetto."



One of the factors that influenced me was the fact that MAGI's *SynthaVision* system did not use polygons. Rather, everything was built from boolean combinations of mathematical primitives, such as ellipsoids, cylinders, truncated cones. As you can see in the illustration, the lightcycles are created by adding and subtracting simple solid mathematical shapes.

This encouraged me to think of texturing in terms not of surfaces, but of volumes. First I developed what are now called "projection textures," which were also independently developed by a quite a few folks. Unfortunately (or fortunately, depending on how you look at it) our Perkin-Elmer and Gould SEL computers, while extremely fast for the time, had very little RAM, so we couldn't fit detailed texture images into memory. I started to look for other approaches.

Noise



The first thing I did in 1983 was to create a primitive space-filling signal that would give an impression of randomness. It needed to have variation that looked random, and yet it needed to be controllable, so it could be used to design various looks. I set about designing a primitive that would be "random" but with all its visual features roughly the same size (no high or low spatial frequencies).

I ended up developing a simple pseudo-random "noise" function that fills all of three dimensional space. A slice out of this stuff is pictured. In order to make it controllable, the important thing is that all the apparently random variations be the same size and roughly isotropic. Ideally, you want to be able to do arbitrary translations and rotations without changing its appearance too much. You can find a C version of my original 1983 code for the first version in Appendix A (actually my first implementation was in FORTRAN).

My goal was to be able to use this function in functional expressions to make natural looking textures. I gave it a range of -1 to +1 (like sine and cosine) so that it would have a dc component of zero. This would make it easy to use noise to perturb things, and simply "fuzz out" to zero when scaled to be small.

Noise itself doesn't do much except make a simple pseudo-random pattern. But it provides seasoning to help you make things irregular enough so that you can make them look more interesting.

The fact that noise doesn't repeat makes it useful the way a paint brush is useful when painting. You use a particular paint brush because the bristles have a particular statistical quality - because of the size and spacing and stiffness of the bristles. You don't know, or want to know, about the arrangement of each particular bristle. In effect, oil painters use a controlled random process (centuries before John Cage used the concept to make post-modern art).

Noise allowed me to do that with mathematical expressions to make textures.

Pixel stream editing

In late 1983 I wrote a language to allow me to execute arbitrary shading and texturing programs. For each pixel of an image, the language took in surface position and normal as well as material ID, ran a shading, lighting and texturing program, and output color. As far as I've been able to determine, this was the first shader language in existence (as my grandmother would have said, who knew?).

Rob Cook at Pixar had, independently, developed an editable expression parser to parse user-defined arithmetic expressions at each surface sample. He called this technique "Shade Trees." But Shade Trees had no notion of flow-of-control (conditionals, variably iterated loops, procedures).

Pat Hanrahan has told me that he got the inspiration to make a full procedural shading language after he visited MAGI and I showed him what you could do by having access to a user-defined language at every pixel. Pat then designed and implemented the

"RenderMan" shading language at Pixar (for which he received a well-deserved Technical Academy Award).

The key leap of faith I made (odd then, obvious now) is that you should just be able to go ahead and run whatever program you feel like at each surface sample, and that it should be easy to keep quickly modifying this program to refine your results. In order to make things run fast, I modified MAGI's existing SynthaVision renderer to create an intermediate file, after the visible surface and normal calculations have been done. The file just contained a stream of pixel samples, each consisting of *{ Point , Normal , SurfaceId }*. I would stream these samples into my procedural shader, which would then spit out a final RGB for each sample. The big advantage of this is that I could keep running the shader over and over, without having to redo the (in 1983) very expensive point/normal calculations.

By far the oddest thing about the environment at MAGI, in retrospect, was the fact that they ran everything in FORTRAN 66. This was a legacy issue - the SynthaVision ray tracer was originally written by Bob Goldstein, one of the founders of MAGI, sometime prior to 1968 (Bob's first paper on doing volumetric booleans by ray tracing, was published in the journal *Simulation* in 1968). Since then it had simply grown in the same language. FORTRAN 66 was very limiting - lacking recursion, insensitive to case, limited in variable name length to six characters or less, and a host of other qualities that reflected the era it came out of - the engineering culture up to the mid-sixties, which was very unlike the more countercultural aesthetic that nurtured UNIX and C at Bell Labs.

I was one of a group of young upstarts at MAGI who were into UNIX and C, but were not permitted to use it, for legacy reasons. So my solution was to build an entire language on top of FORTRAN. I implemented in FORTRAN only those core "kernel" functions that needed to be computed quickly, such as Noise, or that needed to use built-in math libraries, such as Sin and Cos. For everything else, I used my homegrown shading language. For this reason, I called it "kpl", for "Kernel Programming Language". It has been claimed that the letters "kpl" could also be interpreted in other ways, but frankly I just don't see it.

Kpl was a special purpose language - the only thing I really cared about was being able to manipulate floating point vectors verey easily. This led to a number of language design decisions which made everything easier. In the next section I'll briefly describe the language.

The important thing about reducing everything to floating point vectors was that I could treat normal perturbation, local variations in specularity, nonisotropic reflection models, shading, lighting, etc. as just different forms of procedural texture - the environment does not make any a priori assumption about these things, so it was easy to mix it up and try different models.

The language for Pixel Stream Editing

The language was very simple, but it got the job done. The important thing was that it compiled immediately into an intermediate P-code, which executed very fast. That allowed me to do fast repeated design iterations, with visual feedback at each iteration.

Perhaps the oddest feature of the language was that every variable maintained a separate stack - so scoping for any variable was based on run-time execution, not lexical. This turned out to be *extremely* useful for procedural texturing, since it provided an easy and flexible way to create nested data environments. The basic features of the language are outlined below:

- Stack language
- Post-process to visible surface algorithm
- Intermediate "point/normal/id list" data-structure
- Evaluated at every surface sample
- Variables set at each sample:
    - Point
    - Normal
    - Id
- All values are vectors of floating point
- Transform matrices are vectors of length 12 or 16
- Values are TRUE iff at least one component is non-zero
- Every variable is a stack of values
- Flow of control:
    - IF THEN ELSE
    - LOOP with CONDITIONAL-BREAK
    - PROCEDURE with ARGS
- Scoping
    - ASSIGN (var is global to this proc)
    - PUSH-ASSIGN (var is local to this proc)
    - POP ON PROCEDURE EXIT
- Library of kernel functions, including:
    - + - * /
    - Index
    - Noise
    - Bias
    - Gain
    - Sin
    - Cos
    - Pow
    - Mul (matrix)
- Library implemented in the language, including:
    - Abs
    - Dot

- Cross
- ...

Experience and interaction

My interactive process when working with this first shader was really simple. I had two interaction windows: a text editor and an rendered-image display. I'd make text modifications, hit a key that would save and run, and then look at the result. Then I'd make more text modifications, etc.

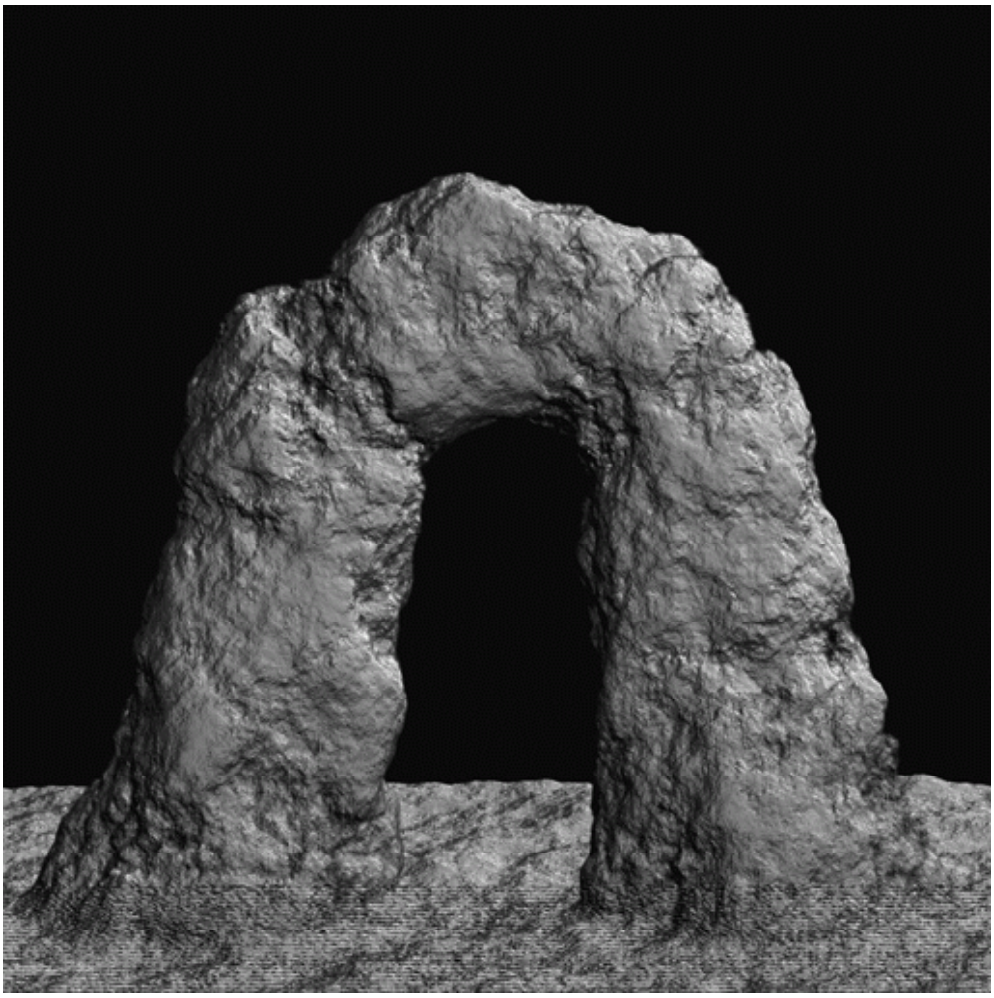
When playing with this interaction environment, I found that I could get big speedups by recomputing just a sub-window in the image where I really wanted to see an effect. I was able to get a good rhythm going of iterative shading/texturing as long as I could see the result within about 15 seconds of hitting the ENTER key (compilation took much less than a second; pretty much all of the time was taken up calculating the image). Of course, back in 1984 this didn't allow me to compute very high resolution images (around then we had only a few Mhz to play with), but any longer than 15 seconds of computation was too long to maintain a good interactive process. In any case, by looking at carefully selected subwindows, I could interactively steer the quality of the complete texture. Ultimately, it turned out to be fairly straightforward to interactively design subtle textures such as the marble vase below, which took about 20 minutes back then to render at high resolution (but would require only a matter of seconds on today's computers):

Industry adoption

I presented this work first at a course in SIGGRAPH 84, and then as a paper in SIGGRAPH 85. Because the techniques were so simple, they quickly got adopted throughout the industry. The release of Pixar's commercial-strength RenderMan language helped a lot. By around 1988 noise-based shaders where *de rigeur* in commercial software.

I didn't patent. As my grandmother would have said...

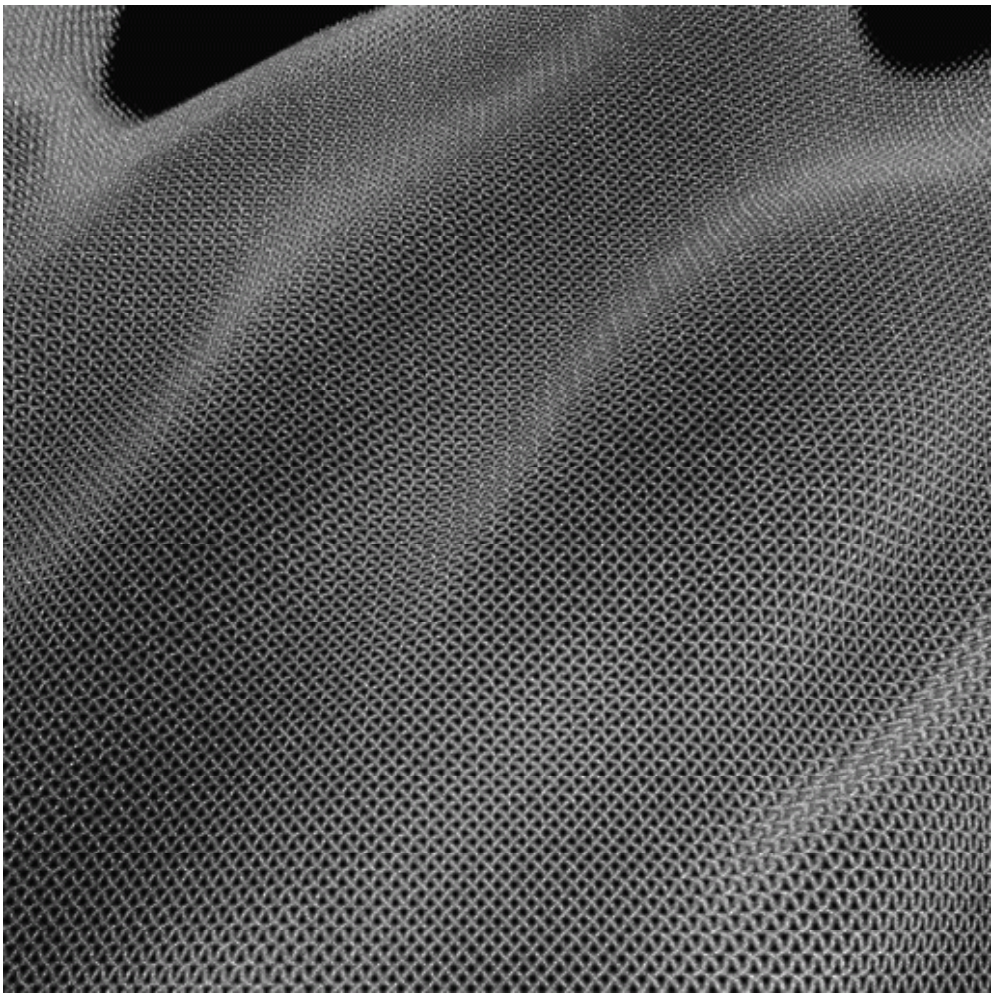
Hypertexture



Meanwhile, I joined the faculty at NYU and did all sorts of research. One of the questions I was asking in 1988 and 1989 was whether you could use procedural textures to unify rendering and shape modeling. I started to define volume-filling procedural textures and render them by marching rays through the volumes, accumulating density along the way and using the density gradient to do lighting.

I worked with a student of mine, Eric Hoffert, to produce a SIGGRAPH paper in 1989 [PERLIN89]. The technique is called hypertexture, officially because it is texture in a higher dimension, but actually because the word sounds like "hypertext" and for some reason I thought this was funny at the time. I offer no redeeming excuse.

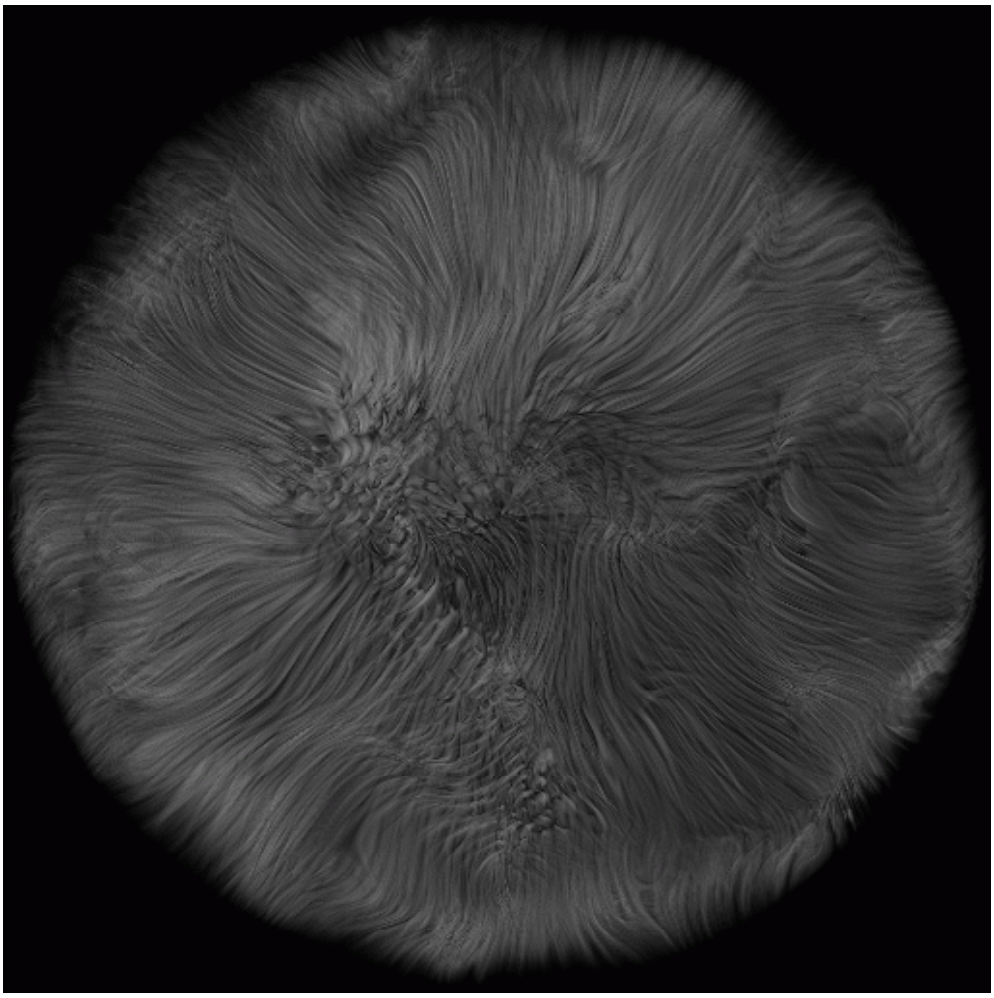
The image above is of a procedurally generated rock archway. Like all hypertextures, it's really a density cloud that's been "sharpened" to look like a solid object. I defined this hypertexture first by defining a space-filling function that has a smooth isosurface contour in the shape of an archway. Then I added to this function a fractal sum of noise functions: at each iteration of the sum I doubled the frequency and halved the amplitude. Finally, I applied a high gain to the density function, so that the transition from zero to one would be rapid (about two ray samples thick). When you march rays through this function, you get the image shown.



I tried to make as many different materials as possible. Above is one of a series of experiments in simulating woven fabric. To make this, I first defined a flat slab, in which density is one when y=0, and then drops off to zero when y wanders off its zero plane. More formally: f(x,y,z) = {if |y| > 1 then 0 else 1 - |y|}.

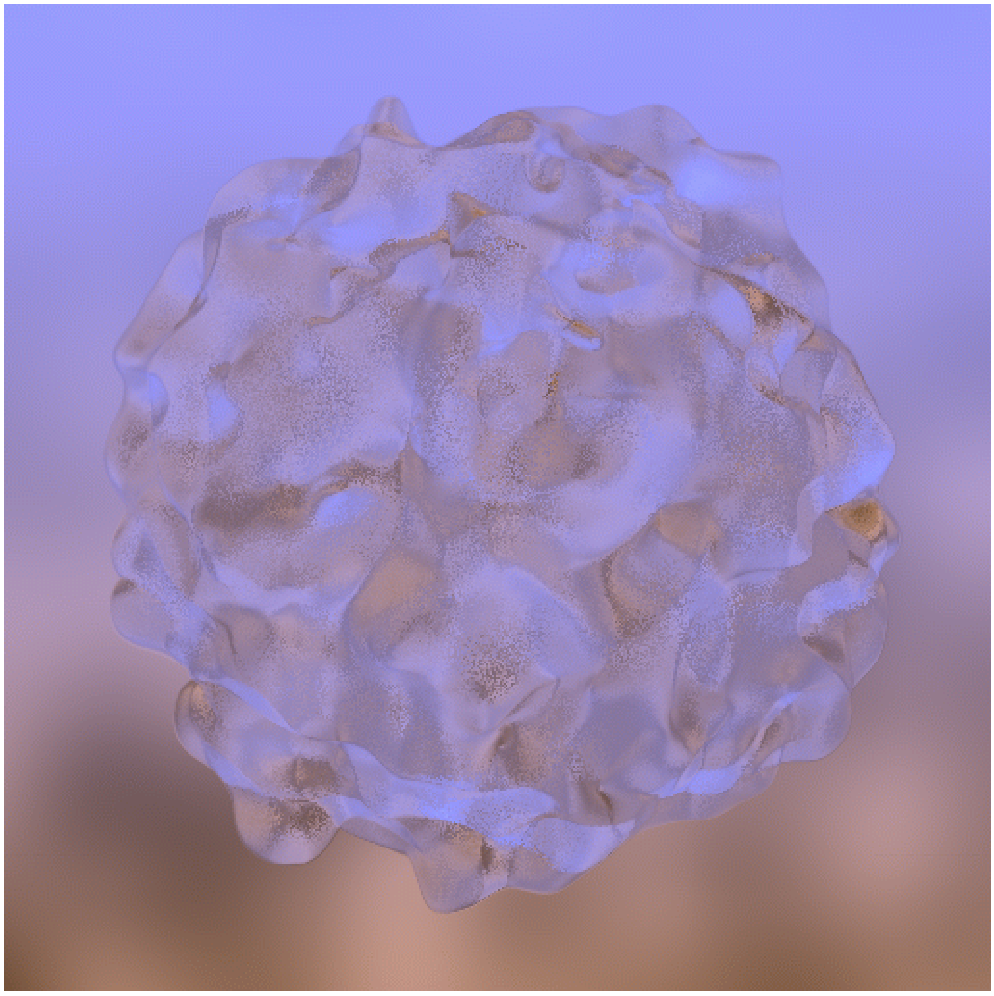
I made the plane ripple up and down by replacing y with y + sin(x)*sin(z) before evaluating the slab function. Then I cut the slab into fibers by multiplying the slab function by cos(x). This gave me the warp threads of a woven material. Finally, I rotated the whole thing by 90^{o} to get the weft threads. When you add the warp and the weft together, you get something like the material on the left.

To make the fiber more coarse (ie: wooley) or conversely more fine, I modulated the bias and gain of the resulting function. To make the surface undulate, I added low frequency noise to y before evaluating anything. To give a nice irregular quality to the cloth, I added high frequency noise into the function.



I also made a Tribble, as shown here, as well as other experiments in "furrier synthesis". Here I shaped the density cloud into long fibers, by defining a high frequency spot function (via noise) onto an inner surface, and then, from any point P in the volume, projecting down onto this surface, and using the density on the surface to define the density at P. This tends to make long fibrous shapes, since it results in equal densities all along the line above any given point on the inner surface.

I made the hairs curl by adding low frequency noise into the domain of the density function. This was the first example in computer graphics of long and curly fur. Around the same time Jim Kajiya made some really cool fur models, although his techniques produced only short and straight fur. Jim had the good sense to use earthly plushy toys for his shape models, instead of alien ones. The earthly ones are more easily recognized by the academy...



It has always seemed to me that there would be advantages in having optical materials with continually varying density, within which light travels in curved paths. The image on the left is a hypertexture experiment in continuous refraction. The object is transparent, and every point on its interior has a different index of refraction. I implemented a volumetric version of Snell's law, to trace the curved paths made by light as it traveled through the object's interior.

The background is not really an out-of-focus scene; it's just low frequency noise added to a color grad. This is a situation in which noise is really convenient - to give that look of "there's something in background, and I don't know what it is, but it looks reasonable and it sure is out of focus."

Meanwhile, back at the Ranch...

Meanwhile, back at the Ranch (if you're reading this you presumably know *which* ranch I'm talking about) the use of noise spread like wildfire. All the James Cameron, Schwartzenegger, Star Trek, Batman, etc. movies started relying on it.

Procedural texture benefits from Moore's law: as computer CPU time becomes cheaper, production companies increasingly have turned away from physical models, and toward

computer graphics. Noise-based procedural shading is one of the main techniques production companies use to fool audiences into thinking that computer graphic models have the subtle irregularities of real objects. For example, Disney put it into their *CAPS* system - you can see it in the mists and layered atmosphere in high end animated features like The Lion King. In fact, after around 1990 or so, *every* Hollywood effects film has used it, since they all make use of software shaders, and software shaders depend heavily on noise. Eventually, they even gave me a Technical Academy Award for it [SCITECH97].

One problem with all this is that as audience expectations improve, the size and computational complexity of shaders has been increasing steadily. For example, "The Perfect Storm" averaged about 200 evaluations of Noise per shading sample. Even with the current impressive performance of computers, each frame took a long time to compute.

Recently I've been working on addressing this problem. In my next chapter, I'll show work I've been doing more recently on making Noise better, faster, and more "hardware friendly".

References:

[EBERT98] *Texturing and Modeling; A Procedural Approach,* Second Edition; Ebert D. et al, AP Professional; Cambridge 1998c;

[PERLIN89] Perlin, K., and Hoffert, E., *Hypertexture,* 1989 Computer Graphics (proceedings of ACM SIGGRAPH Conference); Vol. 23 No. 3.

[PERLIN85] Perlin, K., *An Image Synthesizer,* Computer Graphics; Vol. 19 No. 3.

[SCITECH97] Technical Aphievemegt Award from the Academy of Motion Picture Arts and Sciences, ``*for the development of Perlin Noise, a technique used to produce natural appearing textures on computer generated surfaces for motion picture visual effects."*

Chapter 5

PixelFlow Shading

Marc Olano

OpenGL Extensions and Restrictions for PixelFlow

Jon Leech
University of North Carolina

April 20, 1998

Abstract

This document describes the extensions to OpenGL supported by the PixelFlow API, restrictions forced by the architecture, and as-yet unimplemented features.

READERS OUTSIDE UNC-CH AND HEWLETT-PACKARD PLEASE NOTE: This is an internal working document. The final implementation may differ substantially.

Contents

List of Tables

1 Introduction

This document describes the *PxGL* graphics API for the UNC/Hewlett-Packard *PixelFlow* [3] architecture. PxGL is based on the OpenGL [1] API with extensions, restrictions, and unimplemented features[1]. Only material which differs between PxGL and a conformant OpenGL implementation is covered; readers are expected to be conversant with OpenGL proper.

PixelFlow has enormous flexibility because almost all stages of the graphics pipeline - transformation, rasterization, and shading - are implemented with user-programmable hardware. In order to exploit this capability in the framework of a traditional graphics API, we have extended OpenGL to specify

- When to **load** and **invoke** application-defined code (rather than built-in functionality, such as rendering lit, Gouraud-shaded triangles).
- Which **stage** of the pipeline to invoke it at.
- What **parameters** to pass when the code is executed.

To optimize performance of OpenGL code on PixelFlow, some architectural details of the machine are exposed to the API. Using these features may relax some OpenGL guarantees or invariants in return for greatly improved performance. They include

- **Primitive and state distribution**, which balances rendering load across the parallel geometry processors while affecting the order in which primitives are composited into the frame buffer.
- **Display list optimization**, which increases performance of upper stages of the pipeline while relaxing knowledge of global state.

While PixelFlow has far more flexibility in most respects than more traditional graphics accelerators, it also has certain constraints not present in those machines. Most notably, the nature of the image-composition architecture forces a *frame oriented* paradigm on the API, and implies that there is no valid frame buffer containing pixel colors until after rasterization and shading of all primitives in that frame is complete. PixelFlow also uses a *deferred shading* model, in which pixel color is not computed until after visibility determination. The consequences of these and other minor architectural and design decisions are that

- Additional, non-standard OpenGL calls are required to delimit the start and end of frame generation.
- Much of the global rendering state (textures, lights, view matrices, and other state which is not associated to individual primitives) must be defined prior to start of frame and may not change within the frame.
- Many API calls are only allowed at specific points in the process of generating a frame.

[1] PixelFlow will support a fully conformant OpenGL API, but in general that mode will not be used because of its expected substantial performance cost.

- Most types of blending and stenciling are not supported, and composition order of primitives is not guaranteed.
- Access to the frame buffer may only take place after end of frame.

Finally, many features of the rich OpenGL API are not implemented in PxGL at this time, though they may be added later.

1.1 Roadmap

The remainder of this document will address the following areas:

- Frame generation (§2).
- Controlling primitive and state distribution (§3).
- Extending the OpenGL namespace (§4).
- Loading application-defined code (§5).
- Programmable rasterization (§6).
- Programmable shading (§7).
- Programmable lighting (§8).
- Used-defined functions (§??).
- Other programmable pipeline stages (§9).
- Transparency and shadows effects (§10).
- Display list optimization (§11).
- Multiple application threads (§12).
- OpenGL variances (§13).
- Unsupported OpenGL features (§14).

1.2 Change Log

This is revision *Revision* : 1.9 of *Source* : */tmp*$_m$*nt/net/hydra/pp0/doc/software/opengl/tex/RCS/pxgl.tex*, *v*. Changes from the next most recent revision are delimited by change bars (or approximations thereof in the HTML version).

Changes in revision 1.9 (July 22, 1997):

- Changed all uses of **glInquireParameterEXT()** to **glGetMaterialParameterNameEXT()** or **glGetRastParameterNameEXT()** as appropriate.

- Note that **glGetLightParameterNameEXT()** and other stage-specific inquiry functions will need to be documented and created.
- Added to section on primitive and state distribution, including **pxDistributionMode()** and **glGenDataEXT()**.
- Added section on user-defined functions.

Changes in revision 1.8 (August 1, 1996):

- Changed references from Division to Hewlett-Packard to reflect PFX sale to HP.
- Added new inquiry calls for rasterizer and shader parameters (though details remain to be documented).
- Rearranged glossary entries in section 7 to group parameter terminology together, at Rich Holloway's suggestion.
- Added section on transparency and blending effects, including **glTransparencyEXT()**.

Changes in revision 1.7 (March 22, 1996):

- **glShaderEXT()** now allows different shaders on front and back faces of primitives.
- Added discussion to **glSurfaceEXT()** definition of the restriction of a single value for uniform and nonvarying parameters, regardless of whether the front or back face of a primitive is being rasterized.
- Added discussion to **glMaterialVaryingEXT()** definition of the reason for the apparently-redundant *shaderid* argument.
- Added **glLightModelEXT()** to lighting chapter, specifying that user-defined shader parameters are handled in the same way as OpenGL material parameters.

Changes in revision 1.6 (February 12, 1996):

- First version released to outside readers; added disclaimers.
- Removed definitions of hardware-specific terms like composition/geometry network parameters, and changed definitions of varying/nonvarying/uniform parameters to eliminate dependence on those terms.
- Added *face* argument to **glSurfaceEXT()**.

Changes in revision 1.5 (December 17, 1995):

- Added calls for light groups and loadable light functions.
- Removed **glGenShaderEXT()** and folded its functionality into **glNewShaderEXT()**.

- Added sections (though little text yet) for atmospheric and image warping shader stages.
- Changed **glSurfaceParamEXT()** to **glRastParamEXT()** to avoid too-close similarity to **glSurfaceEXT()**.
- Updated to reflect separate-namespace model for parameters and separation of instance and current values. In particular, **glBindParameterEXT()** has been replaced by **glSurfaceEXT()**, although the name of the latter may change.
- Rewrote interpolator introduction.

Changes in revision 1.4 (November 14, 1995):

- Moved document from LaTeX 2.09 to LaTeX 2_ε.
- Added changebars using changebar.sty.

Changes in revision 1.3 (November 11, 1995):

- Added flat interpolator for per-primitive constant parameters.
- Added **glBindParameterEXT()** and **glGetParameterEXT()**.
- **glShaderEXT()** now takes a face argument. Added `GL_FRONT_SHADER_EXT` and `GL_BACK_SHADER_EXT` as targets to **glGet()**.
- Worked on definitions of composition network and geometry network parameters; more work is needed.

2 Frame Generation

The underlying hardware model in OpenGL is that primitives are specified by the application and immediately drawn - vertices are transformed and lit, rasterization and texturing are done, and final pixel colors are copied into the frame buffer, or blended with existing frame buffer contents. Global parameters affecting transformation, rasterization, and shading of primitives, such as the projection matrix, light bindings, blending modes, and so on, may be changed at any time.

This model is not compatible with PixelFlow's image composition and deferred shading paradigms. In order to achieve good performance on the machine, the API must be *frame-oriented*; that is, it must specify several *stages* in the process of generating a frame, and different types of OpenGL operations may occur only during specific stages. The stages and the types of calls that may take place during them are:

- **Frame setup** - establish viewing, lighting, and shading parameters that will apply throughout the frame.
- **Geometry definition** - traverse the database, rasterizing primitives.
- **End of frame** - perform image composition, shade pixels, and read/write directly to the frame buffer.

2.1 Frame Setup

The setup stage begins by calling **glBeginFrameEXT()**. In this stage, parameters which globally affect the scene are defined. This includes defining the projection matrix, loading light functions, creating lights and light groups, changing light source parameters, loading shader functions, creating shaders, changing nonvarying shader parameters, loading rasterizer functions, binding textures, and any other operations that must be known before primitives can be rasterized and shaded (a complete list of OpenGL calls and the stages they may be called for is in section 13). Parameters of the scene such as the viewport size, antialiasing kernel, and background color are also set here; these must be known to define the *rendering recipe*.

PxGL currently allows only a single projection matrix to be used during a frame. Many lighting environments may be used, but they must be defined as *light groups*. Many textures may be used, but they must be defined during frame setup using the *texture object* calls[2].

2.2 Geometry Definition

The geometry stage begins by calling **glStartGeometryEXT()**. In this stage, primitives are defined and rasterized by different *rasterizer boards*. Valid calls include operations on the modelling and texture matrices, setting material values and other attributes, changing the current texture, and other changes to global state which affect only transformation and rasterization. Display lists may be compiled and executed, or primitives may be issued in immediate mode.

2.3 End of Frame

The final stage begins when **glEndFrameEXT()** is called. Once it returns, the frame buffer is defined. At this time it may be accessed using functions like **glReadPixels()** or **glCopyTexture()**[3]. We expect to support other frame buffer operations such as **glAccum()** at a later date.

2.4 Example

This code fragment draws a frame containing a single red triangle. Lights are assumed to have been defined previously.

```
glBeginFrameEXT();

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 3.0);
```

[2]The reason for these restrictions is that while performing deferred shading, the viewing, lighting, and texturing environment is assumed to be the same for all samples. If this were not the case, such information would have to be encoded along with each sample, which would enormously increase the amount of pixel memory needed for a sample. By creating named objects representing these environments, we regain this capability, although not at OpenGL's per-primitive granularity.

[3]Hopefully, for e.g. shadow maps.

```
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.0, 0.0, -2.0);

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glStartGeometryEXT();

    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_TRIANGLES);
        glVertex3f(-1.0, -1.0, 0.0);
        glVertex3f( 0.0,  1.0, 0.0);
        glVertex3f( 1.0, -1.0, 0.0);
    glEnd();

glEndFrameEXT();
```

Example - Frame generation

3 Controlling Primitive and State Distribution

The PixelFlow architecture achieves scalability by using many parallel *rasterizers*, each of which is responsible for transforming and rasterizing a portion of the database, and *shaders*, each of which is responsible for lighting and shading a portion of the pixels in the image. However, primitives are defined in sequential order by the application. So to achieve good rasterization performance, all the primitives defined in the course of a frame must be *distributed* among the rasterizers.

PxGL has a built-in distribution algorithm, and in most cases, an application does not need to be aware of or make changes in this algorithm. However, in some cases application performance can be increased by modifying how primitives are distributed.

This section describes how primitives are distributed, the implications of the distribution algorithm on graphical state maintenance and performance, and how applications may control distribution.

3.1 Primitive Distribution Algorithm

In the remainder of this section, we assume that a PixelFlow system with N rasterizer boards is being used, and that M geometric primitives are to be distributed, where $M \gg N$.

To be done: call to specify processor groups + comments on ordering implications of distributing primitives, state maintenance, per-vertex state not neccessarily affecting global state.

The calls controlling distribution are[4]

`GLenum pxDistributionMode(GLenum` *`type`*`, GLenum` *`mode`*`, GLint` *`param`*`)`

[4]The pbase headers don't use GL types for the prototypes, and return void - this inconsistency needs to be resolved.

Changes how GL commands are distributed to rasterizers and shaders. *type* specifies the type of commands to be affected, and may take on the following values:

> `PX_PRIMITIVE_EXT` - affects sequences of commands delimited by a **glBegin()** . . . **glEnd()** block, which are normally rasterizer primitives such as triangles.
>
> `PX_STATE_EXT` - affects all other commands not within a block[5].
>
> `PX_TEXTURE_EXT` - affects textures[6].

mode specifies how that type of command is distributed, and may take on the following values:

> `PX_DEFAULT_EXT` - commands are sent in according a default mapping scheme.
>
> `PX_BROADCAST_EXT` - commands are sent to all rasterizers that may use them.
>
> `PX_ROUND_ROBIN_EXT` - commands are sent to a single rasterizer or shader, but successive commands are sent to different rasterizers or shaders in a simple sequence specified by *param*, for load balancing purposes.
>
> `PX_ROUND_ROBIN_WEIGHTED_EXT` - commands are sent to a single rasterizer or shader, but successive commands are sent to different rasterizers or shaders in a sequence determined by the cost of the commands, for load balancing purposes[7].
>
> `PX_SPECIFIED_GPS_EXT` - commands are sent to a set of rasterizers and shaders specified by *param*[8].

param controls details of the distribution. For `PX_ROUND_ROBIN_EXT` mode, it is the *blocking factor* - *param* commands are sent to each rasterizer or shader before shifting to the next. For `PX_SPECIFIED_GPS_EXT` mode, it is the rasterizer to send commands to. *param* is currently ignored for the other modes.

`GL_INVALID_ENUM` is generated if *type* or *mode* are not one of the allowed values.

`GL_INVALID_VALUE` is generated if *param* is less than 1 (for `GL_ROUND_ROBIN_EXT` mode) or an invalid rasterizer or shader ID (for `GL_SPECIFIED_GPS_EXT` mode).

`GLenum pxGetDistributionMode(GLenum` *`type`*`, GLenum *`*`mode`*`, GLint *`*`param`*`)`

Returns the distribution *mode* and *param* used for the specified *type* of command.

This call may not be placed in a display list.

`GL_INVALID_ENUM` is generated if *type* is not one of the valid command types passed to **pxDistributionMode()**.

[5] Not implemented; may never be implemented

[6] Which commands are "textures", exactly?

[7] How might this be parameterized?

[8] Eventually, *param* will specify a *processor group ID*, referring to an arbitrary set of processors established with other pxgl calls. At present, it is just a rasterizer number, with rasterizers numbered starting at 0.

4 Extending the OpenGL Namespace

The C language binding of OpenGL [2] includes several namespaces: *functions*, *types*, and *enumerants*. PxGL extends the function and enumerant namespaces and adds several new namespaces: *shader parameters*, *shader functions*, *light parameters*, *light functions*, *rasterizer parameters*, *rasterizer functions*, and *interpolators*. Examples of these namespaces are given.

In accordance with the ARB[9] guidelines for extensions to OpenGL, all additions to the existing namespaces are postfixed by **EXT** for functions and **_EXT** for enumerants.

4.1 Functions

The function namespace refers to C calls made by an application, such as **glBegin()** and **glEnable()**. About 20 new calls are introduced in PxGL, such as **glStartGeometryEXT()** and **glShaderEXT()**. New calls are discussed in detail elsewhere in this document.

4.2 Enumerants

The enumerant namespace refers to compile-time integral constants used to denote options, values, flags, and other parameters to API functions. PxGL adds enumerants for the new calls it introduces, such as `GL_ALL_PRIMITIVES_EXT` (an allowed parameter to the function **glMaterialInterpEXT()**). PxGL also allows some existing functions to *accept* additional enumerant values in the context of extensions, such as passing an enumerant denoting a user-defined sphere rasterizer to **glBegin()** (which normally accepts only enumerants corresponding to the primitives defined in OpenGL). Finally, some existing functions will *generate* or *return* new enumerant values, such as `GL_UNSUPPORTED_OPERATION_EXT` (which may be generated by calling functions in unsupported modes, and later returned by **glGetError()**).

4.3 New Namespaces

Application-defined code may be inserted at many stages of the graphics pipeline, primarily for rasterization, surface shading, and lighting. To call this code and pass appropriate values to it, several new namespaces are introduced corresponding to the various types of code and parameters.

Because such code (with the exception of built-in functionality like triangle rasterizers or the OpenGL shading model) is not known at compile time, a way to dynamically define the namespaces is needed. This is accomplished by functions which map from ASCII string **names** of code and parameters to numeric **identifiers**[10] which are passed to PxGL calls[11].

The new namespaces and the sections in which their uses are discussed are

- Rasterizer functions and parameters, and parameter interpolators (§6).

[9] OpenGL Architecture Review Board.

[10] Should generated IDs be `GLenum` or `GLuint`? Adding enumerants at runtime is of questionable legality; using integers causes incompatibilities with existing calls like **glMaterial()**.

[11] It would be possible to pass names everywhere and avoid this mapping, at enormous performance cost.

- Shader functions, instances, and parameters (§7).
- Light functions, instances, and parameters (§8).
- Atmospheric functions and parameters (§9.1).
- Image manipulation functions and parameters (§9.2).

4.3.1 Names of OpenGL Objects

OpenGL parameters such as light and material properties are given string names (§15). There are unique parameter IDs corresponding to the different parameters, such as ambient light color and ambient surface color. This differs from OpenGL, where the same *pname*, such as `GL_AMBIENT`, may be used to refer to both light and material properties. For backwards compatibility, the OpenGL IDs are accepted as aliases of the actual parameter IDs.

Stuff to be done...

- Querying instance/global, interpolator, and default value for shader parameters
- Built-in shader function, shader parameters (also for rasterizers, lights, etc.)
- Specifying transformation of parameters (also for rasterizers, lights, etc.)
- Talk some more about the parameter namespaces and how they relate to OpenGL *pnames*.

5 Loading Application-Defined Code

Adding application-defined code written in the PixelFlow *shading language* [5] to the PxGL graphics pipeline is done at runtime[12].

The application identifies such code using string *names* that symbolically refer to different modules; the API hides details of how the names are mapped into object files which are loaded into the hardware[13]. For example, a light function using the Torrance-Sparrow model might be named `torrance`; a sphere rasterizer function might be named `sphere`; and a marble shader function might be named `marble`.

Application-defined code may be loaded using this call:

`GLenum glLoadExtensionCodeEXT(GLenum` *stage* [14]`, const GLubyte *`*name*`)`

[12] The mechanism used involves compiling code in the *shading language* into shared object files that are loaded on demand.

[13] Although we can expect that the name will either be a Unix filename component, or a key to look up a filename.

[14] Do we want to load code for different stages with a single interface? We distinguish between stages with **glGetMaterialParameterNameEXT()** and **glGetRastParameterNameEXT()** for example.

Loads application-defined code for the specified pipeline *stage* identified by *name*. Returns an enumerated `id` which is passed to other calls controlling when the code is to be used.

May be called with a built-in function or called again for application-defined code that's already been loaded. No action is taken, but the same valid `id` is returned.

stage may take on the following values:

`GL_LIGHT_FUNCTION_EXT` - load a light function. `id` is passed to **glNewLightEXT()**.

`GL_RASTERIZER_FUNCTION_EXT` - load a rasterizer function. `id` is passed to **glBegin()**.

`GL_SHADER_FUNCTION_EXT` - load a shading function. `id` is passed to **glNewShaderEXT()**.

`GL_ATMOSPHERIC_FUNCTION_EXT` - load an atmospheric function. `id` is passed to[15].

`GL_WARPING_FUNCTION_EXT` - load an image warping function. `id` is passed to[16].

`GL_INVALID_ENUM` is generated if *stage* is not one of the allowed values, and 0 is returned.

`GL_INVALID_VALUE` is generated if *name* does not exist, and 0 is returned.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**, and 0 is returned.

Code loaded with **glLoadExtensionCodeEXT()** usually has associated *parameters*; rasterizers may also have associated *interpolators*. Loading code may have the side effect of extending those namespaces. At present, there is a single namespace for parameters even though they are accessed by different calls depending on the stage in which those parameters are used. Thus, we require user-defined namespace scoping to distinguish both the stage and the specific object within that stage which the parameter applies to; for example, `rast_sphere_radius` and `shader_polkadot_radius`[17].

To map parameter names into identifiers, use the calls **glGetMaterialParameterNameEXT()** or **glGetRastParameterNameEXT()**.

6 Programmable Rasterization

The programmable rasterization model used in PxGL extends the **glBegin() / glEnd()** mechanism used to define built-in primitive types such as triangles and lines. These new terms are introduced:

[15] Yes, to what?
[16] And again, to what?
[17] We should recommend namespace conventions.

Interpolator - A method for combining parameter values specified at one or more discrete locations on a primitive being rasterized to generate values for that parameter at all other locations on the primitive where it is not specified. The most common interpolators are named `constant` (corresponding to flat shading on a primitive), `flat` (corresponding to **glShadeModel(**`GL_FLAT`**)**, e.g. flat shading on individual polygons within a primitive), and and `linear` (corresponding to **glShadeModel(**`GL_SMOOTH`**)**, e.g. Gouraud shading on polygons within a primitive). Other interpolator types may be defined for user-specified rasterizer functions.

Since interpolation considered as a mathematical process is tightly bound to the geometrical definition of a surface, most interpolators are only defined for specific types of primitives. Interpolators have string *names* and corresponding enumerated *parameter IDs*, referred to as `interpname` and `interpid` in code examples

Rasterizer Function - A function which takes as input a set of *rasterizer parameters* and generates screen-space samples at which the function is visible. A rasterizer function represents a type of geometric primitive; its parameters determine a specific instance of that geometry. In abstract terms, the function creates geometry, transforms it according to the current model-view and projection matrices, and samples it. At visible samples, *shader parameters* defined for the current shader are computed using a specified *parameter interpolator* and copied into the *sample buffer*.

Rasterizer Parameter - A parameter to a rasterizer function. Some examples include vertices of polygons, sphere radii, or control points of parametric patches.

Sequence Point - Specifies the binding time for a group of rasterizer and shader parameters. A *rasterizer function* may require one or more sequence points to define a specific instance of its geometry. In many cases, including all the OpenGL primitive types, the *rasterizer parameters* bound at the sequence point will simply be vertices of a surface. Other examples include center and radii of spheres, twist vectors of Hermite patches, or coefficients of general quadric surfaces[18].

6.1 Loading and Using Rasterizer Functions

To use an application-defined rasterizer function, the following steps must be taken:

- Load the rasterizer function and obtain its ID with **glLoadExtensionCodeEXT()**
- Obtain parameter IDs of rasterizer parameters using **glGetRastParameterNameEXT()**.
- Call **glBegin()** with the rasterizer ID to start delimiting sequence points of a rasterizer function.
- Specify rasterizer parameters using **glRastParamEXT()** and bind them using **glSequencePointEXT()**.

[18] Rasterizer writers will have to document which parameters are per-block and which are per-sequence-point.

- Call **glEnd()** to finish delimiting sequence points of the function and call the rasterizer function.

6.1.1 Example

In the following example, a rasterizer function named `spheres` is loaded. The function has two parameters, the `center` and `radius` of the sphere; each sequence point defines a separate sphere. Two unit-radius spheres which touch at the origin and are centered at (1,0,0) and (-1,0,0) are drawn.

```
// Load the rasterizer and obtain its ID
GLenum spherefuncid =
    glLoadExtensionCodeEXT(GL_RASTERIZER_FUNCTION_EXT, "spheres");

// Obtain IDs for named parameters
GLenum centerid = glGetRastParameterNameParameterEXT("rast_sphere_center");
GLenum radiusid = glGetRastParameterNameParameterEXT("rast_sphere_radius");

glBeginFrameEXT();
    glStartGeometryEXT();

    GLfloat vertminus[3] = { -1, 0, 0 };
    GLfloat vertplus[3] = { 1, 0, 0 };

    // Draw the two spheres
    glRastParamfEXT(radiusid, 1.0);
    glBegin(spherefuncid);
        glRastParamfvEXT(centerid, &vertminus);
        glSequencePointEXT();

        glRastParamfvEXT(centerid, &vertplus);
        glSequencePointEXT();
    glEnd();
```

Example - Using rasterizer functions

6.2 Rasterizer API Definitions

There is currently an naming inconsistency where some calls use **RastParam** and others use **RastParameter**. This should be resolved, probably in favor of the latter.

`void glGetRastParamEXT(GLenum` *`paramid`*`, TYPE *`*`params`*`)`

Returns the value of the specified parameter in *params*.

`GL_INVALID_ENUM` is generated if *paramid* is not a valid rasterizer parameter.

`GLenum glGetRastParameterNameEXT(GLchar *name_string)`

Returns the parameter ID corresponding to the string *name*.

`GL_INVALID_NAME_STRING_EXT` is generated if *string* is not a parameter of any rasterizer, and 0 is returned.

`GLchar * glGetRastParameterStringEXT(GLenum pname)`

Returns the string name corresponding to the specified parameter ID.

`GL_INVALID_ENUM` is generated if *pname* is not a valid parameter ID, and `NULL` is returned.

`void glSequencePointEXT()`

Binds parameters of the rasterizer and shader functions in use.

`GL_INVALID_OPERATION` is generated when **glSequencePointEXT()** is called other than between **glBegin()** and **glEnd()**.

`void glRastParamEXT(GLenum paramid, TYPE params)`

glRastParam assigns values to rasterizer parameters. *paramid* specifies which parameter will be modified. *params* specifies what value or values will be assigned to the parameter.

`GL_INVALID_VALUE` is generated if *paramid* is not a defined rasterizer parameter ID.

6.3 glVertex() and Sequence Points

Vertices defining built-in primitive types are rasterizer parameters. The following two code sequences have identical effects:

```
glVertex3f(x,y,z);
```

Defining a vertex using glVertex()

```
GLenum vertid = glGetRastParameterNameEXT("gl_vertex");
GLfloat point[4] = { x, y, z, 1.0 };
...
glRastParamfvEXT(vertid, &point);
glSequencePointEXT();
```

Defining a vertex using rasterizer extensions

6.4 Vertex Array Extensions for Rasterizers and Shaders

These will be needed, but can't be finalized until the GL 1.1 specification is out.

6.5 Interpolators

Every rasterizer function has one or more interpolators associated with its geometry, which take shader parameters specified at control points and generate parameter values at all samples. All rasterizers may use the *constant* interpolator, which copies a single value into all samples. Rasterizers defined by OpenGL all support the *flat* interpolator, which copies a separate constant value into each successive primitive (triangle, line segment, quadrilateral, etc.) in a group, and the *linear* interpolator, which fits a linear function (possibly perspective-corrected) to the first two or three vertices of a primitive.

There is also an *implicit* interpolator, which ignores parameter values specified at sequence points. Its exact function varies depending on the rasterizer and parameter type. For built-in rasterizers, the implicit interpolator can only be applied to texture coordinates, implementing the functionality of **glTexGen()**.

Other types of rasterizers may use these interpolators, if they make sense, or define new interpolators corresponding to their geometry[19]. For example, a triangle with 3 additional sequence points at the midpoints of its edges might define a *quadratic* interpolator, to allow smoother shading between triangles. A parametric patch might define an interpolator which applies the same weights to shader parameters as to control points. A sphere or general quadric surface rasterizer might interpret the *implicit* interpolator to generate texture coordinates and normals based on the intrinsic geometry of the surface.

6.6 Interpolator API Definitions

`void glGetMaterialInterpEXT(GLenum` *`paramid`*`, GLenum` *`primtype`*`, GLenum *`interpid`*`)`

> Returns the interpolator used for rasterizing the specified shader parameter for the specified primitive type.
>
> `GL_INVALID_ENUM` is generated if *paramid* is not a valid shader parameter or if *primtype* is not a valid primitive type.

`void glMaterialInterpEXT(GLenum` *`paramid`*`, GLenum` *`primtype`*`, GLenum` *`interpid`*`)`

> Sets the *interpolator* to be used for rasterizing the specified shader parameter for the specified primitive type. A primitive type is required because most interpolators are defined only for specific types of geometry.
>
> *interpid* is usually an interpolator ID for a specific primitive. Five interpolators are built-into PxGL:
>
> `GL_IMPLICIT_INTERPOLATOR_EXT` is implemented for texture coordinates in built-in rasterizers, according to the **glTexGen()** parameters[20]. When rasterizing user defined primitives, it is intended to allow generating normals and texture coordinates based on the intrinsic geometry of the object.
>
> `GL_CONSTANT_INTERPOLATOR_EXT` copies the parameter value current when

[19] We don't have a way to get IDs for interpolators loaded as part of rasterizers, yet - something like a **glGetInterpolatorNameEXT()** call is needed.

[20] Do we want to implement it for surface normals, too?

glBegin() is called into all samples rasterized for that primitive or group of primitives. It is guaranteed to be implemented for all primitive types and all parameter types.

`GL_FLAT_INTERPOLATOR_EXT` copies the parameter value current when the last vertex or sequence point defining a primitive is called into all samples rasterized for that primitive. Unlike the constant interpolator, a group of primitives defined in a **glBegin() / glEnd()** block may have a different value specified for each primitive. This corresponds to **glShadeModelEXT(**`GL_FLAT`**)**.

`GL_LINEAR_INTERPOLATOR_EXT` is implemented for all built-in primitive types and parameters, and corresponds to **glShadeModel(**`GL_SMOOTH`**)**[21].

`GL_DEFAULT_INTERPOLATOR_EXT` is a way to specify the most "natural" type of interpolator for a primitive; linear for a polygon, implicit for a sphere, bicubic for a patch, and so on.

primtype is either a valid primitive type or the special value `GL_ALL_PRIMITIVES_EXT`. In the latter case, only `GL_CONSTANT_INTERPOLATOR_EXT`, `GL_FLAT_INTERPOLATOR_EXT`, or `GL_DEFAULT_INTERPOLATOR_EXT` may be specified.

`GL_INVALID_ENUM` is generated if *paramid* is not a valid shader parameter, if *primtype* is neither a valid primitive type nor `GL_ALL_PRIMITIVES_EXT`, or if *interpid* is not a valid interpolator.

`GL_INVALID_OPERATION` is generated if *interpid* is not defined for the specified *paramid* and *primtype*.

To be added: glGenDataEXT() and **glDeleteDataEXT()**.

7 Programmable Shading

The programmable shading model used in PxGL is based on the RenderMan [4] shading language, but use of some terms differ and these new terms are introduced:

Shader Function - A function, either built-in to PxGL or loaded at runtime, which takes as input a set of *shader parameters* and generates as output a color. A shader function is conceptually applied to each sample of a primitive which was rasterized with a corresponding *shader* applied[22]. Shader functions have string *names* and corresponding enumerated IDs, referred to as `shaderfunc` and `shaderfuncid` in code examples.

Shader - An instance of a shader function which binds a subset of the function's parameters to be *nonvarying* for all samples to which the shader is applied. This is done primarily to increase rasterization and shading speed and to reduce traffic on the PixelFlow image composition network. Shaders have enumerated IDs, referred to as `shaderid` in code examples.

[21] Note that in PxGL, interpolation is applied to shading parameters *before* lighting, rather than to color *after* lighting, as in OpenGL. This allows true Phong shading, avoiding the artifacts caused by OpenGL's Gouraud interpolation of Phong-lit vertices.

[22] Deferred shading means that in practice, only samples which affect visibility are actually shaded.

Shader Parameter - An input argument to a shader function. These fall into three types depending on how they arrive at the shading hardware: *uniform*, *nonvarying*, and *varying* parameters. Shader parameters have string *names* and corresponding enumerated IDs, referred to as `paramname` and `paramid`[23] in code examples.

Nonvarying Parameter - A shader parameter whose value is the same for all samples rasterized using that shader. A non-*uniform* parameter of a *shader function* may be chosen to be either nonvarying or *varying* on a per-*shader* basis using **glMaterialVaryingEXT()**.

Uniform Parameter - A shader parameter whose value is the same for all samples rasterized using that shader. Uniform parameters cannot be made *varying*[24].

Varying Parameter - A shader parameter whose value may be different in each sample rasterized using that shader.

7.1 Creating Shaders

To create a shader, the following steps must be taken:

- Load a shader function and obtain its ID with **glLoadExtensionCodeEXT()**.
- Create the new shader and obtain a shader ID using **glNewShaderEXT()**.
- Obtain parameter IDs of shader parameters using **glGetMaterialParameterNameEXT()**.
- Specify which shader parameters are varying using **glMaterialVaryingEXT()** (all parameters not otherwise specified are assumed to be uniform).
- Instantiate the shader with **glEndShaderEXT()**.

After creating the shader, nonvarying parameter values may be set using **glSurfaceEXT()**. These parameter values can be changed at any time before start of geometry.

7.1.1 Example

This code fragment loads a hypothetical shader function named `phong_shader`. The shader function has two parameters, named `gl_shader_color` (intrinsic color) and

[23] OpenGL uses *pname* to refer to material parameters such as emission color, which are shader parameters of the builtin OpenGL shading model. This discrepancy should be resolved; Rich suggests an explanation of parameter *names* vs. parameter *IDs*.

[24] The distinction between uniform parameters and nonvarying parameters is subtle from the user's point of view, and these definitions need work: both are sent to the shader GPs over the geometry network, but uniform parameters are held on the GP during shading code execution, while nonvarying parameters are copied into pixel memory. The distinction is primarily an efficiency measure to reduce composition network bandwidth requirements.

`gl_shader_normal` (surface normal)[25]. Two shaders are created. The first, `phongshader`, allows both color and normal to vary. The second, `redshader`, has a nonvarying intrinsic color of red.

```
// Load the named shader and obtain its ID
GLenum phongfuncid =
    glLoadExtensionCodeEXT(GL_SHADER_FUNCTION_EXT, "phong_shader");

// Obtain IDs for named parameters
GLenum colorid = glGetMaterialParameterNameEXT("gl_shader_color");
Glenum normalid = glGetMaterialParameterNameEXT("gl_shader_normal");

// Create a shader with ID 'phongshader', allowing both parameters to vary
GLenum phongshader = glNewShaderEXT(phongfuncid);
    glMaterialVaryingEXT(phongshader, colorid);
    glMaterialVaryingEXT(phongshader, normalid);
glEndShaderEXT();

// Create 'redshader', allowing only normals to vary and
//  binding the nonvarying color to red.
GLfloat red[3] = { 1, 0, 0 };
GLenum redshader = glNewShaderEXT(phongfuncid);
    glMaterialVaryingEXT(redshader, normalid);
glEndShaderEXT();
glSurfacefvEXT(redshader, colorid, &red);
```

Example - Creating shaders

7.2 Using Shaders

To use a shader once it has been created, the following steps must be taken:

- Select the shader using **glShaderEXT()**.
- Specify the interpolation method to be used for *varying* shader parameters using **glMaterialInterpEXT()**.
- Define a primitive, setting values of varying shader parameters using **glMaterial()**.

7.2.1 Example

This continues the previous example, defining three triangles. The first uses `redshader` to draw a red phong-lit triangle with linearly interpolated normals. The second uses `phongshader` to draw a vertex-colored triangle using linear interpolation of the vertex colors. The third uses `phongshader` to draw a green triangle using constant interpolation.

[25] Note that these parameters are also parameters of the built-in OpenGL shader; they are used by the loadable shader so the example can make shortcut calls like **glNormal()** and **glColor()** to specify shader parameters, rather than **glMaterial()**.

```
// Select the red-colored shader
glShaderEXT(GL_FRONT_AND_BACK, redshader);

// Choose a linear interpolator for normals and draw a red
//  phong-shaded triangle.
glMaterialInterpEXT(normalid, GL_TRIANGLES, GL_LINEAR_INTERPOLATOR_EXT);

glBegin(GL_TRIANGLES);
    for (i = 0; i < 3; i++) {
        glNormal3fv(normal[i]);
        glVertex3fv(vertex[i]);
    }
glEnd();

// Select the phong shader, use linear interpolation for color,
//  and draw a vertex-colored phong-shaded triangle
glShaderEXT(GL_FRONT_AND_BACK, phongshader);

glMaterialInterpEXT(colorid, GL_TRIANGLES, GL_LINEAR_INTERPOLATOR_EXT);

glBegin(GL_TRIANGLES);
    for (i = 0; i < 3; i++) {
        glColor3fv(color[i]);
        glNormal3fv(normal[i]);
        glVertex3fv(vertex[i]);
    }
glEnd();

// Change to constant interpolation for color, and draw a green
//  phong-shaded triangle.
glMaterialInterpEXT(colorid, GL_TRIANGLES, GL_CONSTANT_INTERPOLATOR_EXT);

GLfloat green[3] = { 0, 1, 0 };
glColor3fv(green);

glBegin(GL_TRIANGLES);
    for (i = 0; i < 3; i++) {
        glNormal3fv(normal[i]);
        glVertex3fv(vertex[i]);
    }
glEnd();
```

Example - Using shaders

There is a subtle difference between the first and third triangles: the first uses a shader where color is *nonvarying*, so that all primitives rendered using that shader will be red. The third triangle uses a shader where color is *varying*, but the constant interpolator causes the

color to be fixed on that particular triangle[26].

7.3 Shading API Definitions

`void glDeleteShaderEXT(GLuint shaderid)`

Removes the definition of the specified shader; *shaderid* is unused after this call.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glEndShaderEXT()`

Instantiates a shader created by **glNewShaderEXT()**. All shader parameters which are not explicitly specified in previous calls to **glMaterialVaryingEXT()** are made *nonvarying*; values of these parameters are set with **glSurfaceEXT()**.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**, or when not preceded by a corresponding **glNewShaderEXT()**.

`void glGet(GLenum pname, TYPE *params)`

glGet() is extended to accept parameters `GL_FRONT_SHADER_EXT` and `GL_BACK_SHADER_EXT`, which return the current front and back face shaders as specified via **glShaderEXT()**.

`void glGetMaterial(GLenum face, GLenum paramid, TYPE *params)`

glGetMaterial() is extended so that *paramid* can refer to shader parameters defined by dynamically loaded shaders.

`GL_INVALID_ENUM` is generated if *paramid* is not a valid shader parameter.

`GLenum glGetMaterialParameterNameEXT(GLchar *name_string)`

Returns the parameter ID corresponding to the string *name_string*.

`GL_INVALID_NAME_STRING_EXT` is generated if *name_string* is not a parameter of any shader, and 0 is returned.

`void glGetMaterialParametersEXT(GLuint shaderid, GLenum *pnames)`

Returns a list of parameter IDs used by the specified shader. *pnames* must have room for at least the number of IDs specified by **glGetNumMaterialParametersEXT()**.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

[26] The purpose of the constant interpolator is to reduce work done during rasterization; it's appropriate when performing (for example) flat shading. The same visual effect could also be achieved by using the linear interpolator and specifying the same color at each vertex, but rasterization speed would be lower.

`GLchar * glGetMaterialParameterStringEXT(GLenum` *`pname`*`)`

Returns the string name corresponding to the specified parameter ID.

`GL_INVALID_ENUM` is generated if *pname* is not a valid parameter ID, and `NULL` is returned.

`GLuint glGetNumMaterialParametersEXT(GLuint` *`shaderid`*`)`

Returns the number of material parameters accepted by the specified shader. Used in conjunction with **glGetMaterialParametersEXT()**.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`void glGetSurfaceEXT(GLuint` *`shaderid`*`, GLenum` *`face`*`, GLenum` *`paramid`*`, TYPE *`params`*`)`

Retrieves the value of a *nonvarying* parameter of the specified shader. Bound values are set by **glSurfaceEXT()**.

`GL_INVALID_ENUM` is generated if *face* is not `GL_FRONT` or `GL_BACK`, or if *paramid* is not a bound parameter of *shaderid*.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`GLboolean glIsMaterialParameterEXT(GLuint` *`shaderid`*`, GLenum` *`pname`*`)`

Returns `TRUE` if *pname* is a parameter of the specified shader, `FALSE` otherwise.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID, and `FALSE` is returned.

`GL_INVALID_ENUM` is generated if *pname* is not a valid parameter ID, and `FALSE` is returned.

`GLboolean glIsMaterialUniformEXT(GLuint` *`shaderid`*`, GLenum` *`pname`*`)`

Returns `TRUE` if *pname* is a *uniform* parameter of the specified shader, `FALSE` otherwise.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID, and `FALSE` is returned.

`GL_INVALID_ENUM` is generated if *pname* is not a valid parameter ID, and `FALSE` is returned.

`GLboolean glIsShaderEXT(GLuint` *`shaderid`*`)`

Returns `TRUE` if *shaderid* is used for an existing shader, `FALSE` otherwise.

`void glMaterial(GLenum` *`face`*`, GLenum` *`paramid`*`, TYPE` *`params`*`)`

glMaterial() is extended so that *paramid* can refer to shader parameters defined by dynamically loaded shader functions.

`GL_INVALID_ENUM` is generated if *paramid* is not a shader parameter either of the built-in OpenGL shading function or of a shader function previously loaded.

`void glMaterialVaryingEXT(GLuint` *`shaderid`*`, GLenum` *`paramid`*`)`

Specifies that a parameter is *varying* for this shader. All parameters of a shader are *uniform* or *nonvarying* unless specified as varying by the time **glEndShaderEXT()** is called[27].

`GL_INVALID_ENUM` is generated if *paramid* is not a valid shader parameter or a *uniform* parameter.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`GL_INVALID_OPERATION` is generated if called other than between **glNewShaderEXT()** and **glEndShaderEXT()**.

`GLuint glNewShaderEXT(GLenum` *`shaderfuncid`*`)`

Creates and returns a shader ID for a new instance of the specified shader function.

`GL_INVALID_ENUM` is generated if *shaderfuncid* does not refer to a valid shader function, and 0 is returned.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**, and 0 is returned.

`void glShaderEXT(GLenum` *`face`*`, GLuint` *`shaderid`*`)`

Sets the shader to be used for shading the specified face of primitives defined following the call. *face* may be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

`GL_INVALID_ENUM` is generated if *face* is not one of the allowed values.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`void glSurfaceEXT(GLunit` *`shaderid`*`, GLenum` *`paramid`*`, TYPE` *`params`*`)`

Sets the value of **nonvarying** parameters of a shader instance. The values of **varying** parameters are set with **glMaterial()**.

Nonvarying parameters cannot be specified separately for front and back faces; there is a single value used regardless of whether the front or back face of a primitive is rasterized. This can be addressed by using different shaders on front and back faces.

A nonvarying parameter has an initial value defined by the shader using that parameter. The value is set when the shader is loaded.

`GL_INVALID_ENUM` is generated if *paramid* does not refer to a nonvarying parameter of the specified shader.

`GL_INVALID_VALUE` is generated if *shaderid* is not a defined shader ID.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

[27] While *shaderid* appears redundant, keeping the parameter allows the possibility of changing a parameter between varying and nonvarying on the fly, in a possible future implementation.

7.4 To Be Done

- Parameter Transformation (normals, texture matrix).
- Parameter Generation (**glTexCoord()**, sphere normals).
- Implicit Parameters (texture scale factors, texture ID, normals).
- `GL_FRONT_AND_BACK` vs. uniform parameters and optimized lists.

8 Programmable Lighting

The programmable lighting model used in PxGL introduces these new terms:

Light Function - A function which takes as input a set of *light source parameters* and a set of *shader parameters* at a sample, and generates an illumination at that sample which is used by a *shader function* to compute color of the sample.

Light Group - A subset of all existing light instances, used to illuminate specified primitives during shading. Only one light group may be active at any time.

8.1 Creating Lights

To create a light, the following steps must be taken:

- Load a light function and obtain its ID with **glLoadExtensionCodeEXT()**
- Create the new light and obtain a light ID using **glNewLightEXT()**.
- Obtain parameter IDs of light source parameters using **glGetLightParameterName**[28]**EXT()**.
- Call **glLight()** to specify light source parameters.

8.1.1 Example

I don't have a good example of a user-defined light function. This example just creates a new instance of the built-in OpenGL light function, which is named `gl_light_function`. The light is made a red, diffuse, infinite light in direction -Z.

```
glBeginFrameEXT();

    // Get the light function ID for the built-in light model
    //  by "loading" it.
    GLenum lightfuncid =
        glLoadExtensionCodeEXT(GL_LIGHT_FUNCTION_EXT, "gl_light_function");

    // Create a new instance of the OpenGL light function
```

[28] This call needs to be added.

```
GLenum lightid = glNewLightEXT(lightfuncid);

// Get IDs of light source parameters. We do not really
//  need to do this for the built-in light function; GL_POSITION
//  and GL_DIFFUSE could be used instead.
GLenum positionid = glGetLightParameterNameEXT("gl_light_position");
GLenum diffuseid = glGetLightParameterNameEXT("gl_light_direction");

GLfloat position[4] = { 0.0, 0.0, -1.0, 0.0 };
GLfloat diffusecolor[4] = { 1.0, 0.0, 0.0, 1.0 };

glLightfv(lightid, positionid, &position);
glLightfv(lightid, diffuseid, &diffusecolor);
```

Example - Creating a light

8.2 Using Lights

There is no limit on the number of lights which may be created (above and beyond the built-in OpenGL lights). Lights are placed in *light groups*, which are arbitrary subsets of the defined lights with enumerated IDs; the *current light group* may be changed at any time and that set of lights is applied when shading primitives. Initially a single light group, `GL_DEFAULT_LIGHT_GROUP_EXT`, exists and is the current light group.

To change the lighting environment, the following steps must be taken:

- Optionally create a new light group.
- Place desired lights in the light group.
- Specify the current light group.
- Render primitives with the specified light group illuminating them.

8.2.1 Example

This continues the previous example, placing the new light in a new light group, selecting that as the current light group, and drawing a triangle.

```
// Create a new light group
GLuint groupid = glNewLightGroupEXT();

// Add the new light to this group
glEnableLightGroupEXT(groupid, lightid);

glStartGeometryEXT();

glLightGroupEXT(groupid);

// Primitives drawn now are lit by the new light
```

Example - Using a light

8.3 Light API Definitions

`void glDeleteLightEXT(GLenum` *`lightid`*`)`

Removes the definition of the specified light; *lightid* is unused after this call.

`GL_INVALID_VALUE` is generated if *lightid* is not a defined shader ID.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glDeleteLightGroupEXT(GLuint` *`groupid`*`)`

Removes the definition of the specified light group; *groupid* is unused after this call.

`GL_INVALID_VALUE` is generated if *groupid* is not a defined light group.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glDisable(GLenum` *`cap`*`)`

`void glEnable(GLenum` *`cap`*`)`

glDisable() and **glEnable()** are extended to operate on light groups. When *cap* is `GL_LIGHT`*i*, the specified built-in light is removed from or added to the current light group[29].

`void glDisableLightGroupEXT(GLuint` *`groupid`*`, GLenum` *`lightid`*`)`

`void glEnableLightGroupEXT(GLuint` *`groupid`*`, GLenum` *`lightid`*`)`

Removes or adds the specified light to the specified light group.

`GL_INVALID_VALUE` is generated if *groupid* is not a valid light group ID or *lightid* is not a valid light ID.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glGet(GLenum` *`pname`*`, TYPE *`*`params`*`)`

glGet() is extended to accept parameter `GL_LIGHT_GROUP_EXT`, which returns the current light group as specified via **glLightGroupEXT()**.

`void glGetLight(GLenum` *`lightid`*`, GLenum` *`paramid`*`, TYPE *`*`param`*`)`

glGetLight() is extended so that *paramid* can refer to light source parameters defined by dynamically loaded light functions.

`GL_INVALID_ENUM` is generated if *lightid* is not a valid light or if *paramid* is not a light source parameter of the light

[29] `GL_LIGHTING` could be implemented as a flag on the entire light group; at present it has no effect.

`void glGetLightFunctionEXT(GLenum` *`lightid`*`, GLenum *`*`lightfuncid`*`)`

Returns in *lightfuncid* the light function used by the specified *light*.

`GL_INVALID_ENUM` is generated if *lightid* is not a valid light.

`GLboolean glIsLightEXT(GLenum` *`lightid`*`)`

Returns `TRUE` if *lightid* is used for an existing light, `FALSE` otherwise.

`GLboolean glIsLightGroupEXT(GLuint` *`groupid`*`)`

Returns `TRUE` if *groupid* is used for an existing light group, `FALSE` otherwise.

`void glLight(GLenum` *`lightid`*`, GLenum` *`paramid`*`, TYPE` *`param`*`)`

glLight() is extended so that *paramid* can refer to light source parameters defined by dynamically loaded light functions.

`GL_INVALID_ENUM` is generated if *paramid* is not a light source parameter either of the built-in OpenGL light function or of a light function previously loaded.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glLightGroupEXT(GLuint` *`groupid`*`)`

Sets the light group to be used for lighting primitives specified following the call.

`GL_INVALID_VALUE` is generated if *groupid* is not a defined light group ID.

`void glLightModelEXT(GLenum` *`pname`*`, TYPE` *`param`*`)`

glLightModel() is extended so that when two-sided lighting is enabled via `GL_LIGHT_MODEL_TWO_SIDE`, it includes all **varying** parameters of the shader being used for a primitive. This allows texture coordinates, texture IDs, and user-defined shader parameters to differ on front and back faces of a primitive.

`GLenum glNewLightEXT(GLenum` *`lightfuncid`*`)`

Creates and returns a light ID for a new instance of the specified light function.

`GL_INVALID_ENUM` is generated if *lightfuncid* does not refer to a valid light function, and 0 is returned.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**, and 0 is returned.

`GLuint glNewLightGroupEXT()`

Creates a new light group and returns the group ID. Initially no lights are in the group; lights may be added with **glEnableLightGroupEXT()**.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

9 Programming Other Pipeline Stages - *to be written*

9.1 Atmospheric

Talk about **glFog()** *here.*

9.2 Warping

To be defined.

10 Transparency and Other Blending Effects

Because PixelFlow is an image composition architecture, in which there is not a single frame buffer during rasterization, the effects possible via blending in OpenGL must be done via alternate methods.

Further discussion about blending across frame boundaries and such will go here later.

10.1 Transparency

Transparent primitives may be handled in one of two ways. The first is screen-door transparency. This supports a limited number of levels of transparency, depending on the number of samples/pixel being rasterized, but is the most general method. The second method is a multipass algorithm which extracts all transparent primitives and renders them properly in sorted order using multiple rendering passes to resolve visibility (*Apgar paper citation goes here*). Unlike alpha blending in OpenGL, neither approach relies on the database being traversed in any particular order.

To use transparent primitives, several steps must be taken:

- Enable transparency on a per-frame basis using **glTransparencyEXT()**.
- Enable transparency on a per-primitive basis using **glEnable()**.
- Specify transparent primitives by defining colors with non-unitary alpha components.

The new calls are:

`void glTransparencyEXT(GLenum` *mode*`)`

Specifies the method by which transparent primitives are rendered. Must be called during the frame setup stage (section 2.1).

mode may take on the following values:

`GL_TRANSPARENCY_NONE_EXT` - transparency is not handled. All primitives are treated as opaque regardless of alpha values.

`GL_TRANSPARENCY_SCREEN_DOOR_EXT` - transparency is done by turning on a fraction of the samples in each pixel corresponding to the alpha value of

that fragment. This is usually the fastest and lowest quality mode.

`GL_TRANSPARENCY_MULTIPASS_EXT` - transparency is done by multipass rendering of potentially transparent primitives. This is usually the slowest and highest quality mode.

`GL_INVALID_OPERATION` is generated if called between **glStartGeometryEXT()** and **glEndFrameEXT()**.

`void glDisable(GLenum` *cap*`)`

`void glEnable(GLenum` *cap*`)`

glDisable() and **glEnable()** are extended to support potentially transparent primitives. When *cap* is `GL_TRANSPARENCY_EXT` and is enabled, primitives may be handled using the transparency mode determined by **glTransparencyEXT()**. When disabled, primitives are treated as opaque regardless of their alpha values.

For maximum performance, `GL_TRANSPARENCY_EXT` should be enabled only when potentially transparent primitives are being rasterized.

10.1.1 Determining Transparency

Determining whether or not primitives are transparent at rasterization time is difficult in a deferred-shading architecture, since user-defined shaders need not have an input parameter analogous to the alpha value used by OpenGL. At present, transparency is only handled for primitives using the built-in OpenGL shader[30].

11 Display List Optimization - *to be written*

- How to specify optimization; types of optimizations.
- Inheriting state from environment for constant-interpolated params, binding at **glBegin()**.
- Interaction with **glShadeModelEXT()**.

12 Multiple Application Threads - *to be written*

Discuss multiple AP contexts, ordering issues, frame synchronization points, global namespaces for lights, shaders, and rasterizers, local (perhaps) namespaces for display lists.

13 OpenGL Variances - *to be written*

Tables of (enumerant,relevant calls) and (call,valid frame stages) will go here.

[30] Is this true? We've gone around on possible approaches to shaders generating transparent samples before, but there has been no resolution yet. What does the current implementation do?

- Depth buffer always enabled.
- Depth function always `GL_LESS`.
- Transparency specially handled (see section 10.1).
- And lots more...

14 Unsupported OpenGL Features - *to be written*

Lee's lengthy document should be referenced here.

15 Function, Enumerant, and Name Tables

Parameters of the built-in light, shader, and rasterizer functions have all been assigned string names which map to enumerated IDs. Existing OpenGL enumerants (such as `GL_AMBIENT` or `GL_LIGHT0`) are recognized as aliases for the actual IDs. String names of built-in parameters, and the corresponding OpenGL enumerants, are listed below.

15.1 Light Function and Parameter Names

There is a single built-in light function corresponding to the OpenGL lighting model, named `gl_light_function`. Table 1 lists parameters of this function, which correspond to OpenGL light source parameters.

String Name	OpenGL ID
gl_light_ambient	GL_AMBIENT
gl_light_diffuse	GL_DIFFUSE
gl_light_specular	GL_SPECULAR
gl_light_position	GL_POSITION
gl_light_spot_direction	GL_SPOT_DIRECTION
gl_light_spot_exponent	GL_SPOT_EXPONENT
gl_light_spot_cutoff	GL_SPOT_CUTOFF
gl_light_constant_attenuation	GL_CONSTANT_ATTENUATION
gl_light_linear_attenuation	GL_LINEAR_ATTENUATION
gl_light_quadratic_attenuation	GL_QUADRATIC_ATTENUATION

Table 1: Built-in light source parameter names

15.2 Rasterizer Function and Parameter Names

Table 2 lists the built-in rasterizer function names and the corresponding OpenGL IDs.

String Name	OpenGL ID
gl_rasterizer_points	GL_POINTS
gl_rasterizer_lines	GL_LINES
gl_rasterizer_line_strip	GL_LINE_STRIP
gl_rasterizer_line_loop	GL_LINE_LOOP
gl_rasterizer_triangles	GL_TRIANGLES
gl_rasterizer_triangle_strip	GL_TRIANGLE_STRIP
gl_rasterizer_triangle_fan	GL_TRIANGLE_FAN
gl_rasterizer_quads	GL_QUADS
gl_rasterizer_quad_strip	GL_QUAD_STRIP
gl_rasterizer_polygon	GL_POLYGON

Table 2: Built-in rasterizer functions

There is a single parameter of built-in rasterizers, named `gl_vertex`. Vertices are normally specified using **glVertex()** rather than **glRastParamEXT()** (§6.3).

15.3 Shader Function and Parameter Names

There is a single built-in shader function corresponding to the OpenGL shading model, called `gl_shader_function`. Table 3 lists parameters of this function and the corresponding OpenGL material parameter names.

15.4 Atmospheric Function and Parameter Names

There is a single built-in atmospheric function corresponding to the OpenGL fog model, called `gl_fog_function`. Table 4 lists parameters of this function and the corresponding OpenGL fog parameter names.

15.5 Interpolator Names

Table 5 lists the built-in interpolator functions which may be used with the built-in rasterizer functions. The **constant** and **implicit** interpolators may also be used with any application-defined rasterizer function.

String Name	OpenGL ID
gl_shader_ambient	GL_AMBIENT
gl_shader_diffuse	GL_DIFFUSE
gl_shader_color	Use **glColor()**
gl_shader_specular	GL_SPECULAR
gl_shader_emission	GL_EMISSION
gl_shader_shininess	GL_SHININESS
gl_shader_textureid	Use texture object calls
gl_shader_normal	Use **glNormal()**
gl_shader_u, gl_shader_v	Use **glTexCoord()**
gl_shader_du, gl_shader_dv	Implicitly generated

Table 3: Built-in material parameters

String Name	OpenGL ID
gl_fog_mode	GL_FOG_MODE
gl_fog_density	GL_FOG_DENSITY
gl_fog_start	GL_FOG_START
gl_fog_end	GL_FOG_END
gl_fog_color	GL_FOG_COLOR

Table 4: Built-in atmospheric parameters

String Name	OpenGL ID
gl_interpolator_implicit	GL_IMPLICIT_INTERPOLATOR_EXT
gl_interpolator_constant	GL_CONSTANT_INTERPOLATOR_EXT
gl_interpolator_flat	GL_FLAT_INTERPOLATOR_EXT
gl_interpolator_linear	GL_LINEAR_INTERPOLATOR_EXT
gl_interpolator_default	GL_DEFAULT_INTERPOLATOR_EXT

Table 5: Built-in interpolator names

15.6 Defined Constants

Table 6 lists manifest constants in PxGL which are not in OpenGL, along with the corresponding commands these constants are used in.

Constant	Associated Commands
GL_ALL_PRIMITIVES_EXT	**glMaterialInterpEXT()**
GL_BACK_SHADER_EXT, GL_FRONT_SHADER_EXT, GL_LIGHT_GROUP_EXT	**glGet()**
GL_DEFAULT_LIGHT_GROUP_EXT	**glLightGroupEXT()**
GL_CONSTANT_INTERPOLATOR_EXT, GL_DEFAULT_INTERPOLATOR_EXT, GL_FLAT_INTERPOLATOR_EXT, GL_IMPLICIT_INTERPOLATOR_EXT, GL_LINEAR_INTERPOLATOR_EXT	**glMaterialInterpEXT()**
GL_ATMOSPHERIC_FUNCTION_EXT, GL_LIGHT_FUNCTION_EXT, GL_RASTERIZER_FUNCTION_EXT, GL_SHADER_FUNCTION_EXT, GL_WARPING_FUNCTION_EXT	**glLoadExtensionCodeEXT()**
GL_TRANSPARENCY_EXT	**glEnable()**
GL_TRANSPARENCY_NONE_EXT, GL_TRANSPARENCY_SCREEN_DOOR_EXT, GL_TRANSPARENCY_MULTIPASS_EXT	**glTransparencyEXT()**
GL_UNSUPPORTED_OPERATION_EXT	many

Table 6: Defined constants

16 Glossary

Interpolator - A method for combining parameter values specified at one or more discrete locations on a primitive being rasterized to generate values for that parameter at all other locations on the primitive where it is not specified.

Light Function - A function which takes as input a set of *light source parameters* and a set of *shader parameters* at a sample, and generates an illumination at that sample which is used by a *shader function* to compute color of the sample.

Light Group - A subset of all existing light instances, used to illuminate specified primitives during shading. Only one light group may be active at any time.

Nonvarying Parameter - A shader parameter whose value is the same for all samples rasterized using that shader.

Rasterizer Function - A function which takes as input a set of *rasterizer parameters* and generates screen-space samples at which the function is visible.

Rasterizer Parameter - A parameter to a rasterizer function.

Sequence Point - Specifies the binding time for a group of rasterizer and shader parameters.

Shader Function - A function, either built-in to PxGL or loaded at runtime, which takes as input a set of *shader parameters* and generates as output a color.

Shader Parameter - An input argument to a shader function.

Shader - An instance of a shader function which binds a subset of the function's parameters to be *nonvarying* for all samples to which the shader is applied.

Uniform Parameter - A shader parameter whose value is the same for all samples rasterized using that shader.

Varying Parameter - A shader parameter whose value may be different in each sample rasterized using that shader.

Rasterizer Boards - Hybrid MIMD/SIMD parallel processors which transform subsets of the primitives making up an image, rasterizing *shader parameters* into local *sample buffers* These buffers are later combined using the image composition network as directed by the rendering recipe.

Rendering Recipe - A list of instructions describing how to combine rasterized *screen regions* containing shading parameters using the image composition network, shade the resulting visible samples, and combine shaded samples into the frame buffer. The rendering recipe is normally defined by state such as viewport size and number of supersamples used for antialiasing.

Sample Buffer - buffers on rasterizer boards which contain samples of locally-visible surfaces and shading parameters for those samples.

17 Credits

The PixelFlow API has developed by discussion among the following people[31]:

Dan Aliaga, Jon Cohen, Lawrence Kestleoot, Anselmo Lastra, Jon Leech, Jonathan McAllister, Steve Molnar, Marc Olano, Greg Pruett, Yulan Wang, and Rob Wheeler (UNC), and Rich Holloway, Roman Kuchkuda, and Lee Westover (HP)

[31] I think this covers everyone who had significant input, but please correct me - JPL.

References

[1] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.1).* Silicon Graphics, Inc., 1995. Unpublished; available at UNC in file:/home/pxfl/doc/software/SGI/glspec.ps

[2] OpenGL Architecture Review Board. *OpenGL Reference Manual.* Addison-Wesley Publishing Company, Inc., 1992.

[3] Steve Molnar, John Eyles, and John Poulton. *PixelFlow: High-Speed Rendering Using Image Composition.* Computer Graphics vol. 26 no. 2, July 1992.

[4] Steve Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley Publishing Company, Inc., 1990.

[5] Marc Olano. *PixelFlow Shading Language.* Unpublished; talk to Marc for a copy.

Index

PixelFlow Shading Language
Marc Olano
Revised: 15 September 1997

1 Overview

The PixelFlow shading language is a special purpose C-like language for describing the shading of surfaces on the PixelFlow graphics system. On PixelFlow, some shading function written in the shading language is associated with each primitive. The shading function is executed for each visible pixel (or sample for antialiasing) to determine its color. The language is based heavily on the RenderMan shading language[1].

2 Data

2.1 Built in types

Only a few simple data types are supported. The simplest type is **void**. As with C, it is only used as a return type for functions that have no return value. There is a floating point type, **float**, used for most scalar values. There is a fixed point type, **fixed**, provided for efficiency. And there are literal strings, useful for print formatting[2]. Note that, unlike RenderMan, the string type is not used as an identifier for texture maps, instead a scalar ID is used.

The **fixed** type has two parameters: the size in bits and an exponent. So it is really a class of types, given as **fixed<size, exponent>**. For exponents between zero and the bit size, the exponent can also be thought of as the number of fractional bits. Note however, that an exponent larger than the size or less than zero is perfectly legal. A two byte integer would be **fixed<16, 0>**, while a two byte pure fraction would be **fixed<16, 16>**. It is possible to translate back and forth between the real value and stored value using these equations:

$$\text{real\_value} = \text{stored\_value}^{-\text{exponent}}$$
$$\text{stored\_value} = \text{real\_value}^{\text{exponent}}$$

However, it is much less confusing to always think of the real value. For example, with a **fixed<8,8>**, never think of the value as 128, instead think 0.5. An unspecified fixed point type can also be used, declared simply as **fixed**, and its size and exponent will be chosen automatically[3].

It is also possible to have arrays of these basic types, declared in a C-like syntax (i.e. **float color[3]**). The declaration **float color[3]**, declares color to be a 1D array of three **float**s, **color[0]**, **color[1]**, and **color[2]**. You can also look at **color** as a variable of type **float[3]**, and an equivalent definition would be **float[3] color**. Note the behavior of mixing these two types of definitions: **float[2][3] color_list**, **float[3] color_list[2]** and **float color_list[2][3]** are all equivalent. As with C, it is not necessary to give all of the indices for an array at once. While **color_list[1][1]** is a **float**, **color_list[0]** and **color_list[1]** are each **float[3]** 1D arrays. Where RenderMan uses separate types for points, vectors, normals, and colors, pfman uses arrays.

2.2 Type attributes

As with RenderMan, types may be declared to be either **uniform** or **varying**. A **varying** variable is one that might vary from pixel to pixel, similar **plural** in MasPar's mpl. A **uniform** variable is one that will not vary from pixel to pixel, similar to **singular** in MasPar's mpl. It deserves mentioning again that declaring a variable to be **varying** does not imply that it will vary, only that it might. If not specified, shader parameters default to **uniform** and local variables default to **varying**.

Variables of the **fixed** type may be declared **signed** or **unsigned**. The size of a fixed point type does not include the extra sign bit added by **signed**. So a **signed fixed<15,0>** takes 16 bits. If not specified, all fixed point variables default to **signed**.

---

[1] Upsill, Steve, The RenderMan Companion, Addison-Wesley, 1990.

[2] As of September 13, 1997, strings for calls to printf are not supported.

[3] As of September 13, 1997, automatic fixed point variables are not supported. The sizes produced by automatic fixed types will have to be pessimistic in their size estimation. Error analysis and explicit fixed point sizes is sure to make better use of memory.

There are a number of additional attributes for shader parameters. One transformation type can be given for any parameter. These are **transform_as_vector**, **transform_as_normal**, **transform_as_point**, **transform_as_plane**, or **transform_as_texture**[4]. A parameter can also be declared to be **unit** if it should be unit length[5]. For example, you might declare a parameter

unit transform_as_vector float v[3];

These attributes only affect what happens to the parameter before it is passed to the shader. They do not affect how the parameter is used inside the shader. For example, a **unit** parameter will not remain unit length. These attributes also cannot be used to distinguish versions of an overloaded function.

2.3 User defined types

Aliases can be defined for types with a C-like **typedef** statement. **typedef** is only legal outside function definitions. The **typedef** statement only provides aliases for types, no distinction is made between equivalent types with different names. The statement

typedef float Point[3], Normal[3];

declares **Point** and **Normal** to both be types which can be used completely interchangably with **float[3]**.

3 Expressions

3.1 Operators

The set of operators and operator precedence is fairly similar to that of C (it was based on a grammar for ANSI C). The full list of operators and their precedence is given in Figure 1.

Operation	Associativity	Purpose
`( )`	—	expression grouping
`++ -- []`	—	postfix increment and decrement, array index
`++ -- - !`	—	prefix increment and decrement, arithmetic and logical negation
`( )`	—	type cast
`^`	left	xor / cross product / wedge product[6]
`* / %`	left	multiplication, division, mod
`+ -`	left	addition, subtraction
`&`	left	bitwise and[7]
`|`	left	bitwise or[8]
`<< >>`	left	shift[9]
`< <= >= >`	left	comparison
`== !=`	left	comparison
`&&`	left	logical and
`||`	left	logical or
`?:`	right	conditional expression
`= += -= *= /= ^=`	right	assignment[10]
`,`	—	expression list

Figure 1. Operator precedence

3.2 Operations on arrays[11]

Operations on arrays are defined as the corresponding vecor, matrix, or tensor operation. The unary operations act on all elements of the array. Addition, subtraction, and assignment require arrays of equal

---

[4] As of March 4, 1995, vectors and points are transformed the same and normals and planes are transformed the same.

[5] As of September 13, 1997, **unit** has no affect (parameters declared **unit** are not normalized).

[6] As of March 4, 1995, none of xor, cross product, or wedge product are implemented.

[7] & only works between identical fixed point types.

[8] | only works between identical fixed point types.

[9] As of September 13, 1997, left and right shift are only implemented for varying integer shift values

[10] As of September 13, 1997, ^= is not implemented

[11] As of September 13, 1997, Array cross product, and inverse do not work.

dimension and do the operation between corresponding elements (i.e. **a + b** gives the standard matrix addition of **a** and **b**). The comparison operations also require arrays of equal dimension, though only == and **!=** are defined.

Multiplication between vectors gives a dot product, between vector and matrix, matrix and vector, or matrix and matrix gives the appopriate matrix multiplication. More generally, multiplication between any two arrays gives the tensor contraction of the last index of the first array against the first index of the second array. In other words, for **float a[3][3][3]**, **float b[3][3][3]** and **float c[3][3][3][3]**,

```
c = a * b;
```

is equivalent to

```
float i, j, k, l;
for(i=0; i<3; i++)
    for(j=0; j<3; j++)
        for(k=0; k<3; k++)
            for(l=0; l<3; l++) {
                c[i][j][k][l] = 0;
                for(m=0; m<3; m++)
                    c[i][j][k][l] += a[i][j][m] * b[m][k][l];
            }
```

Division can also be used as a matrix inverse. **1 / a** is the inverse of a square matrix **a** and **b / a** multiplies **b** by the inverse of square matrix **a**.

Finally, the ^ operator gives the cross product between two vectors or the tensor wedge product between two arrays.

3.3 Inline arrays[12]

C-style array initializers are allowed in any expression as an anonymous array. So a 3x3 identity matrix might be coded as **{{1,0,0},{0,1,0},{0,0,1}}**, while the computed elements of a point on a paraboloid might be filled in with **{x, y, x*x+y*y}**.

3.4 Einstein summation notation[13]

Inside any statement block, the uniform integer variables **$1**, **$2**, … are automatically defined. For example for **float a[3], b[3]**, the expression **a[$1] * b[$1]** is equivalent to **a[0]*b[0] + a[1]*b[1] + a[2]*b[2]** (which in this case, is equivalent to **a * b**).

4 Statements

4.1 Compound statements

As with C, anywhere a statement is legal, a compound statement is legal as well. A compound statement is just a list of statements delimited by **{** and **}**.

4.2 Expression statements

Any expression followed by a **;** is a legal statement.

4.3 Standard control statements

Most of the control statements are borrowed directly from C.[14]

if (condition_expression) statement_for_true
if (condition_expression) statement_for_true **else** statement_for_false
while (condition_expression) loop_statement
do loop_statement **until** (condition_expression)**;**
for (initial_expression**;** condition_expr**;** increment_expression) loop_statement
break;
continue;

---

[12] As of September 13, 1997, inline arrays can only have constants for their array elements.
[13] As of September 13, 1997, Einstein summation notation is not implemented
[14] Due to limitations of PixelFlow, the condition_expression's must be **uniform** for all of the looping control statements. The condition for an **if** can be either **uniform** or **varying**.

return;
return return_value_expression**;**

In addition, there are several control statements taken from the RenderMan shading language to aid in shading. They are **illuminance**, **illuminate**, and **solar**.

The **illuminance** statement,

illuminance () statement
illuminance (position_expression) statement
illuminance (position_expression**,** axis_expression**,** angle_expression) statement

acts like a loop over the available light sources. It can also be thought of as an integral over the incoming light. For each light that can hit a pixel at the given position, or can hit a surface at the given position with the given orientation and visibility angle, the light source function is run, returning a light color and intensity that can be used in the statement. The light direction can be accessed using the **px_rc_l** parameter to the shader. The light color can be accessed using the **px_rc_cl** parameter to the shader.

The **illuminate** and **solar** statements,

illuminate (position_expression) statement
illuminate (position_expression**,** axis_angle**,** angle_expression) statement
solar (axis_angle**,** angle_expression) statement
solar ( ) statement

provide the information the **illuminace** statement uses to tell if a light source function should be run or not. They can also be thought of as conditional statements that only execute the associated statement if the current pixel position falls within the light's area. The four statements above correspond to a point light, a spot light, a directional light, and an ambient light[15].

4.4 Declaration statements

Variable declarations can occur anywhere a statement can. They consist of a type and a list of new variable names to declare. Each variable name can have additional array dimensions and an expression for the initial value.

float a[3], b=2*x, c;

declares **a** as an uninitialized 1D **float** array with 3 elements, **b** as a **float** with an initial value twice whatever is in the **x** variable at the declaration time, and **c** as an uninitialized **float**.

Each compound statement defines a new scope, so variables can be redefined within a compound statement without conflicting with function or variable names in other scopes. It is illegal, however, to have a variable in any scope with the same name as any user defined type. This is true even if the **typedef** occurs after the variable declaration.

5 Functions

5.1 Overloading

Function overloading similar to C++ is supported. So functions of the same name that can be destinguished by their input parameters are considered distinct. This provides the ability to have seperate versions of functions for **uniform** and **varying** parameters, **float** and **fixed**, or different fixed point types. Note that functions cannot be overloaded based on their return parameters and operator overloading is not supported.

5.2 Definition

A function definition gives the return type, name, parameters, and body that define the function. Function definitions cannot be nested. By default, function parameters and return types are **uniform**. A simple function definition:

```
float factorial(float n) {
  if (n > 1)
    return n * factorial(n);
```

[15] I don't really like the way this works in RenderMan. Is there a use to placing some of the light code within an **illuminate** statement and some outside? Is it too specialized for a couple of particular light types? Whether I understand it or not, it's there.

```
    else
        return 1;
}
```

The formal parameters to a function have their own scope level between the global scope and the function body, so their names can hide the global function names. As with variables, it is illegal to have a function or parameter with the same name as a user defined type, regardless of where in the source the **typedef** occurs.

5.3 Shading functions[16]

There are several special return types to indicate that a function has some special rendering purpose and may need to be called by the PixelFlow rendering library. These are **primitive**, **interpolator**, **surface**, **light**, and **image**. A **primitive** function computes which pixels are in some rendering primitive like a triangle or sphere; an **interpolator** function computes the value for some shading parameter across a number of pixels; a **surface** function computes the shading on a surface (the archetypal shading function); a **light** function computes the color and intensity of a light; and an **image** computes the final color and location of the image pixels (handling image warping, fog effects, etc.). For all of these functions, each parameter can have a default value in case the graphics library is not given a value for that parameter. These are given just by putting an **= value** (just like variable initialization) in the parameter list. These default values must be compile-time constants. It is perfectly legal to call a surface shading function from inside another surface shading function[17]. In this case, only one illuminance statement can occur in either the original surface shader or any called by it.

5.4 Prototypes

Any function that is to be used before it is defined, or that is defined in a different source file, must have a prototype. A function prototype is just like a function definition, but with a **;** instead of the function body

```
float factorial(float n);
```

5.5 Internal details and External linkage

The **pfman** shading language compiler turns shading language source code into C++ source code that must be further compiled with a C++ compiler. The function definitions created by the compiler and function calls made by it correspond directly to C++ function definitions and function calls. It is possible (and supported) to call C++ functions from shading language functions and to call shading language functions from C++. This facility is limited to functions using types that the shading language supports.

Pfman adds some additional arguments added by the compiler. The new first argument is a pointer to the PixelFlow IGCStream where the instruction stream for the pixel processors should go. The new second argument is a pointer to a PixelFlow GLStage class, which contains information about the rendering context. The new third argument is a pointer to the PixelFlow pixel memory map class. For functions with a varying return value, the new third argument is the address for the return value. All the other arguments follow. There are C++ classes for varying float and fixed parameters giving their address, and in the case of fixed parameters, their size and binary point position. Details of these types and the prototypes for the different kinds of shading functions are beyond the scope of this document.

Standard C or C++ functions can be used by pfman by prefixing their prototype with **extern "C"** or **extern "C++"**. All of the uniform math library routines are declared this way. These tell pfman not to add the extra function parameters. Similarly, pfman functions that contain only uniform operations can be declared **extern "C"** or **extern "C++"** for use by code outside of pfman.

---

[16] As of September 13, 1997, only surface and light are supported.

[17] As of September 13, 1997, it is not possible to call either surface shaders from inside surface shaders.

Implementing PixelFlow Shading

Marc Olano

In the previous sections of this chapter, we covered the interface seen by both application and shader writers. In this section, we cover the basic knowledge of the PixelFlow hardware required to understand the implementation issues. For more details on the PixelFlow architecture, see [Molnar91][Molnar92][Eyles97]. We also cover some intermediate levels of abstraction between PixelFlow and an abstract graphics pipeline and explain how our procedural stages fit into the real PixelFlow pipeline.

Our abstract pipeline consists of procedures for each stage in the rendering process. Since these can be programmed completely independently, it is possible (and expected) that a particular hardware implementation may not have procedural interfaces for all stages. While PixelFlow is theoretically capable of programmability at every stage of the abstract pipeline, our implementation only provided high-level language support for surface shading, lighting, and primitives. The underlying PixelFlow software includes provisions for programmable testbed-style atmospheric and image warping functions, but we did not supply any special-purpose language support for these.

1. High-level view

PixelFlow consists of a host workstation, a number of rendering nodes, a number of shading nodes, and a frame buffer node. The hardware and lower level software handle the scheduling and task assignment between the nodes, so we can consider the flow of data in the system as the pipeline shown in Figure 1. This view is based on the passage of a single displayed pixel through the system. Neighboring pixels may have been operated on by different physical nodes at each stage of this simplified pipeline. This will be covered in more detail later in this chapter. For the purposes of mapping the abstract pipeline onto PixelFlow, the simplified view of the physical PixelFlow pipeline is sufficient.

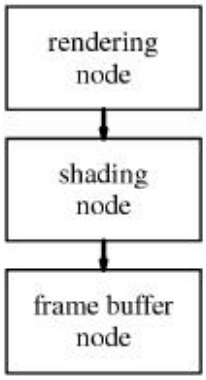


Figure 1. Simplified view of the PixelFlow system

1.1. Applying the abstract pipeline

The mapping of an abstract pipeline onto PixelFlow is shown in Figure 2. This abstract pipeline is divided into stages based on a set of logical rendering tasks. Contrast this with the abstract model presented later in Chapter 8, in which a single shader spans several computational units.

The modeling, transformation, primitive, and interpolation stages are handled by the rendering node. The shading, lighting, and atmospheric stages are handled by the shading node. Finally, the image warping stage is handled by the frame buffer node.

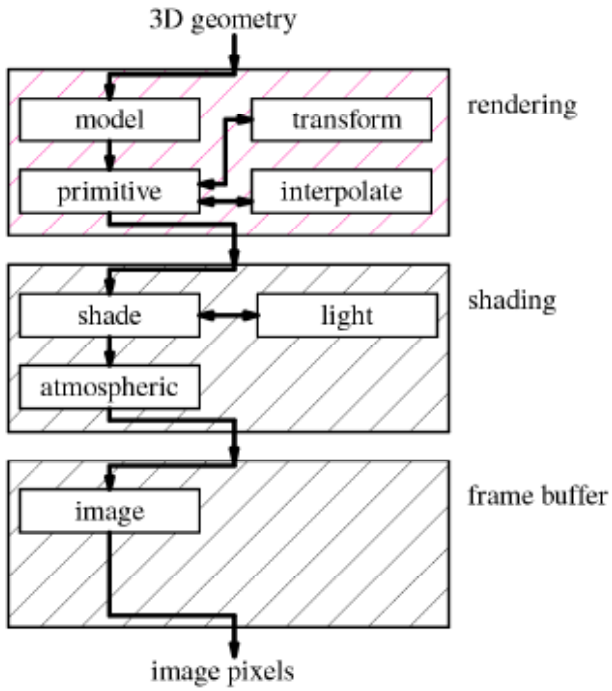


Figure 2. Procedure pipeline.

When mapping the abstract pipeline onto PixelFlow, we maintain the interfaces to the pipeline stages. Thus, the procedures written for PixelFlow should look exactly the same as the procedures written for a different machine with a different organization. The code for each stage is written just as if it were part of some arbitrary rendering system implementing the abstract pipeline.

It is important to notice that the abstract pipeline only provides a conceptual view for programming the stages. It allows the procedure programmer to pretend that the machine is just a simple pipeline instead of a large multicomputer. The real stages do not need to be executed strictly in the order given (and, in fact, are not). The user writing code for one of the stages does not need to know the differences between the execution order given in the abstract pipeline and the true execution order. The mapping of the abstract pipeline onto PixelFlow exhibits several different forms of this.

The first example is the overall organization of the processes on PixelFlow. PixelFlow completes all of the modeling, transformation, primitives, and interpolation in the rendering nodes before sending the shading parameters for the visible pixels on to a shading node. PixelFlow then completes all of the shading, lighting, and atmospheric

effects before sending the completed pixels on to the frame buffer node for warping. On a different graphics architecture, it might make more sense to complete all of the stages for every pixel in a primitive before moving on to the next primitive. Either choice appears the same to users who write the procedures. The abstract pipeline does not include information about the stage scheduling to allow just such implementation flexibility.

The procedures running on the PixelFlow rendering nodes provide another example. The abstract pipeline presents transformation, primitive, and interpolation as if they were a sequential chain of processes. On PixelFlow, the primitive stage drives transformation and interpolation. A procedural primitive function is invoked for each primitive to be rendered. This function calls both transformation and interpolation functions on demand as needed. The results stored for each pixel include its depth, an identifier for which procedural shader to use and the shading parameters for that procedural shader. Once again, the user writes procedures as if they were independent sequential stages and is not aware of the true ordering within the PixelFlow implementation.

The final example is with the shading and lighting stages. The abstract pipeline presents shading and lighting as if the shading stage called the lighting stage for each light. On PixelFlow, the linkage between these stages is not as direct. These two stages run with an interleaved execution scheduled by the PixelFlow software system. This interleaving is explained in more detail in [Olano98]. And again, the interleaved scheduling is hidden from anyone who writes a shading or lighting procedure.

1.2. Parameter manager

Supporting this pipeline is a software framework that handles the details of the rendering process and the communication between the programmable procedures. That communication is assisted by a global parameter manager, implemented on PixelFlow by Rich Holloway. The parameter manager allows each node in the system to find values or pixel memory addresses of the parameters. It also keeps track of other attributes of each parameter - its type and size, default values, whether it needs to be transformed (and how), etc. Whenever a procedure is compiled, an extra *load function* is generated. This load function is run when the procedure is loaded by the application. The load function registers all of the parameters used or produced by the procedure. The parameter manager collects this information and makes sure each parameter is available when it is needed. This global parameter space is similar to the shared memory "blackboard" idea used by MENV [Reeves90].

2. Low-level view

The PixelFlow system data-flow was show in Figure 1. A view of the hardware at that level was sufficient to understand how the abstract pipeline maps onto PixelFlow. We must delve deeper to understand some of the issues that impacted our implementation. Where Figure 1 showed only a single stage for rendering and shading, PixelFlow may have many nodes (see Figure 3). There are also two networks connecting the nodes in the PixelFlow system, the geometry network and composition network. The rendering nodes and shading nodes are identical, so the balance between rendering performance and shading performance can be decided on an application by application basis. The frame buffer node is also the same, though it includes an additional *daughter card* to produce

video output.

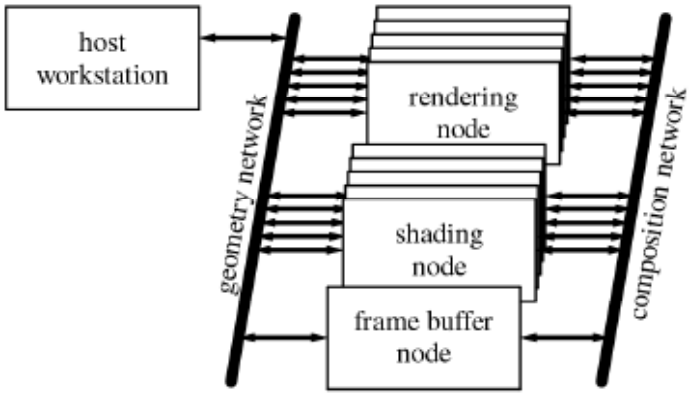


Figure 3. PixelFlow machine organization.

Each rendering node is responsible for rasterizing an effectively randomly chosen subset of the primitives in the scene. The rendering nodes work on one 128x64 pixel *region* at a time (or 128x64 image samples when antialiasing). Many of our examples and tests are based on either an NTSC video screen size of 640x512 pixels with four samples per pixel, or a high-resolution screen size of 1280x1024 pixels. There are 40 regions in an NTSC image with no antialiasing. With antialiasing using four samples per pixel, the NTSC image has 160 regions. Without antialiasing, the high-resolution image also has 160 regions. Therefore, our target is to be able to handle 160-128x64 regions at NTSC video rates of 30 frames per second.

Since each rendering node has only a subset of the primitives, a region rendered by one node will have holes and missing polygons. The different versions of the region are merged using *image composition*. PixelFlow includes a special high-bandwidth network called the *composition network* with hardware support for these comparisons. As all of the rendering nodes simultaneously transmit their data for a region, the network hardware on each node compares, pixel-by-pixel, the data it is transmitting with the data coming in from the upstream nodes. It keeps only the closest of each pair of pixels to send downstream. By the time all of the pixels reach their destination, one of the shading nodes, the composition is complete.

Once a shading node has received the data, it does the surface shading for the entire region. The technique of shading after the pixel visibility has been determined is called *deferred shading* [Deering88][Ellsworth91]. Deferred shading only spends time shading the pixels that are actually visible, and allows us to do shading computations for many more pixels in parallel. With non-deferred shading, each primitive is shaded separately. With deferred shading, all primitives in a region that use the same procedural shader can be shaded at the same time.

In a PixelFlow system with n shading nodes, each shades every n^{th} region. Once each region has been shaded, it is sent over the composition network (without compositing) to the frame buffer node, where the regions are collected and displayed.

3. PixelFlow node

The nodes on PixelFlow all look quite similar (See Figure 4). Each node of the PixelFlow system has two RISC processors (HP-PA 8000's), a 128x64 custom SIMD array of pixel processors, and a texture memory store. Only the rendering nodes make use of the second RISC processor, where the primitives assigned to the node are divided between the processors. The existence of the second RISC processor does not impact our implementation, so we can take the simplified view that there is only one processor on the node and let the lower level software handle the scheduling between the physical processors. The RISC processors share 128 MB of memory, while each pixel processor has access to 256 bytes of local memory. The texture memory exists in several replicated banks for access speed, but the apparent size is 64 MB.

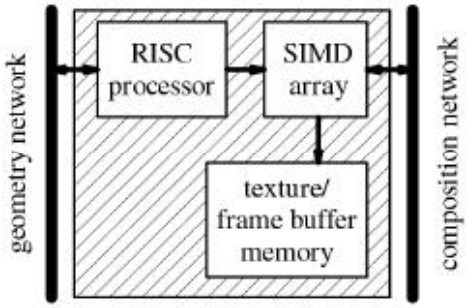


Figure 4. Simple block diagram of a PixelFlow node

Each node is connected to two communication networks. The geometry network, carries information about the scene geometry and other data bound for the RISC processors. This network is four bytes wide and operates at 200 MHz. It can simultaneously send data in both directions, giving a total bandwidth of 800 MB/s in each direction. The composition network handles transfers of pixel data from node to node. It also operates in simultaneously in both directions at 200 MHz. However, the composition network is 32 bytes wide, giving a bandwidth of 6.4 GB/s in each direction. Four bytes of every transfer is reserved for the pixel depth, reducing the effective bandwidth to 5.6 GB/s.

3.1. Compiler target

Every procedural stage on PixelFlow has a testbed-style interface, which allows new stage procedures to be created using the internal libraries of the PixelFlow system. Writing code new procedures using this interface requires a deep understanding of the implementation and operation of PixelFlow, more than will be provided in this dissertation. We provide a high-level, special-purpose language so the users who write new procedures will not need to have that level of understanding of PixelFlow. It also makes rapid prototyping and porting procedures to other systems possible.

The compiler for our special-purpose language produces C++ code that exactly conforms to the testbed interface. This code consists of two functions, a load function (mentioned in section 1.2), and the actual code for the procedure. The code for the procedure is run on the RISC processor and includes embedded *EMC functions*. Each EMC function puts one SIMD instruction into an instruction stream buffer. The EMC prefix that appears on all of these functions stands for *enhanced memory controller*, from the Pixel-Planes SIMD array's origin as a processor-enhanced memory; we use it here just to identify the functions that generate the SIMD instruction stream.

When the C++ code for a procedure is run, the result is a buffer full of instructions for the SIMD array. This instruction stream buffer can be sent to the SIMD array several times without requiring the original C++ code to be re-executed.

There are two forms of EMC function used in PixelFlow. The form used on the shading nodes checks the available space in the instruction stream buffer with each instruction and can re-allocate the buffer on the fly. The form used in the rendering nodes requires a buffer of sufficient size to be allocated at the beginning of the procedure. The reason for this difference, and the issues that result, are discussed in Section [Olano99].

Chapter 6

Procedural Solid Texturing

John C. Hart

Proc. Eurographics/SIGGRAPH Graphics Hardware Workshop, Aug. 1999, pp. 45-53.

Antialiased Parameterized Solid Texturing Simplified for Consumer-Level Hardware Implementation

John C. Hart, Nate Carr, Masaki Kameya

Washington State University

Stephen A. Tibbitts, Terrance J. Coleman

Evans and Sutherland Computer Corp.

Abstract

Procedural solid texturing was introduced fourteen years ago, but has yet to find its way into consumer level graphics hardware for real-time operation. To this end, a new model is introduced that yields a parameterized function capable of synthesizing the most common procedural solid textures, specifically wood, marble, clouds and fire. This model is simple enough to be implemented in hardware, and can be realized in VLSI with as little as 100,000 gates.

The new model also yields a new method for antialiasing synthesized textures. An expression for the necessary box filter width is derived as a function of the texturing parameters, the texture coordinates and the rasterization variables. Given this filter width, a technique for efficiently box filtering the synthesized texture by either mip mapping the color table or using a summed area color table are presented. Examples of the antialiased results are shown.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture --- Graphics processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism --- Color, shading, shadowing and texture.
Keywords: antialiasing, hardware. procedural texturing, solid texturing.

1. INTRODUCTION

Peachey [1985] and Perlin [1985] introduced procedural solid texturing as a method for simulating the sculpture of objects (of arbitrary detail and genus) out of a solid material such as wood or stone, and also the simulation of the natural elements of fire, water (waves), air (clouds) and earth (terrain and planets). Figure 1 through Figure 6 illustrate the variety of images that can be synthesized using procedural solid textures.

Addresses: WSU, School of EECS, Pullman. WA 99164-2752 {hart,ncarr,mkameya}@eecs.wsu.edu.
E&S (Seattle), 33400 8th Ave. S. #136, Federal Way, WA 98003 {stibbitt,tcoleman}@es.com.

Solid texturing creates the illusion that a shape is carved out of a solid three-dimensional substance. The details of a solid texture align across edges and corners of an object surface. For example the grain features on the teapots in Figure 1 and Figure 2 align with the block of material out of which they were sculpted. Depending on the detail and genus of the object, similar alignment of 2-D image texture maps can be very tricky [Peachey, 1985].

Procedural textures require much less memory than stored image textures, and unlike image textures their resolution depends only on computation precision. The sky and water in Figure 3 extend to infinity with non-repeating procedural detail. The fire in Figure 4 is procedurally textured on a single polygon. Zooming into the coastlines of the planet in Figure 5 reveals an arbitrarily intricate level of detail depending on the number of noise functions used in its generation. Figure 6 simulates the reflection of the moon on water without ray tracing or environment mapping by clever manipulation of the color maps of a procedural texture.

While this popular, powerful and flexible technique is found in nearly all high-quality photorealistic rendering packages, it has not yet found its way into consumer-level hardware for real-time rendering. Procedural solid textures would greatly enrich the quality of some of the 2D-image-textured graphical elements found in 3-D interactive games and virtual worlds, not only with wooden and stone objects, but with expansive terrain, oceans and skies filled with non-repeating detail.

Hardware implementation would also support the real-time animation of procedural textures. Varying the parameters of a procedure yields a dynamic animated texture. Depending on the paths chosen through parameter space, these animations can smoothly loop or be non-repeating. These animated textures would support such effects as ripples forming in marble, fire exploding, waves gently rising and falling, clouds billowing, and continents forming on planets.

1.1. Previous Work

Some have identified memory bandwidth as a major obstacle in increasing the performance of real-time graphics hardware. While memory size grows at a rate of 50% per year (one thousandfold over the past two decades), memory bandwidth only grows 12% per year (only tenfold over the past two decades) [Torborg & Kajiya, 1996]. Texture mapping in particular relies heavily on memory, and the bandwidth of this memory is the primary factor limiting the number and complexity of 2-D image textures

Figure 1: Carved wooden teapot.

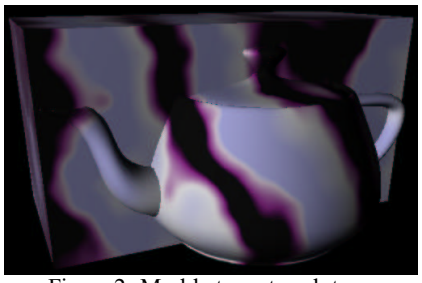


Figure 2: Marble teapot sculpture.

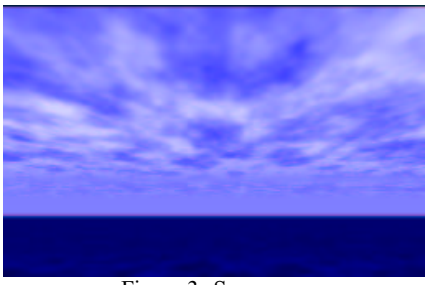


Figure 3: Seascape.

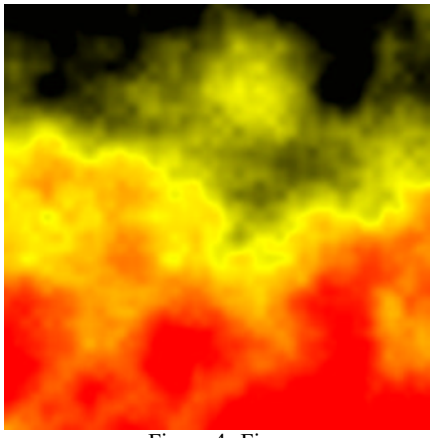


Figure 4: Fire.

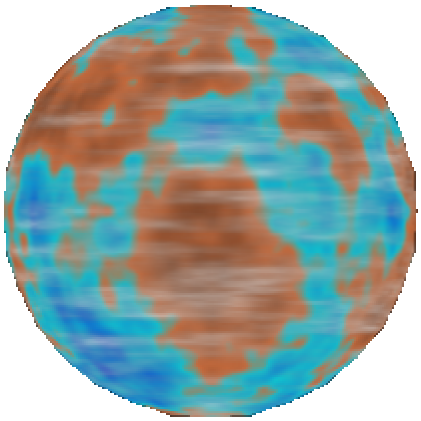


Figure 5: Planet.

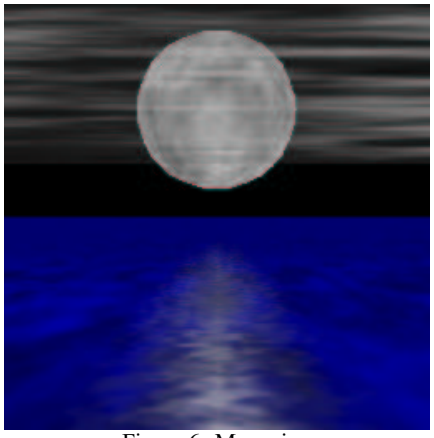


Figure 6: Moonrise.

available in real-time. Some have overcome the memory bandwidth limitation at the expense of increasing memory size to hold multiple redundant copies of the texture [Akeley, 1993], [Montrym, *et al.,* 1997]. Others relaxed the memory bandwidth limitation by reducing the size of the textures via compression [Torborg & Kajiya, 1996],[Beers, *et al.,* 1996]. Procedural texturing hardware is a way of increasing the performance of current graphics hardware by augmenting its existing pre-stored 2-D image textures with a variety of procedural solid textures without impacting the hardware's memory requirements.

Accessing a procedural texture requires more time than an image texture as the texture value must be computed instead of accessed from memory. Hence, real-time procedural texturing has previously only been available in high-end parallel graphics systems. For example, Pixel Planes [Rhoades, *et al.*, 1992], PixelFlow [Molnar, *et al.,* 1992] and the Pixel Machine [Potmesil & Hoffert, 1989] all supported real-time procedural texturing. Indeed, PixelFlow now has a fully-developed procedural shading system, including support for procedural solid texturing [Olano & Lastra, 1998].

Solid texturing is also not new to hardware implementation. The Reality Engine, for example, has the memory bandwidth necessary to support prestored solid texture volumes up to a maximum resolution of 256 x 256 x 64 texture elements [Akeley, 1993]. The InfiniteReality graphics system [Montrym, *et al.,* 1997] has 1GB of physical texture memory that could be organized into a 1024^3 pre-stored solid texture volume.

Antialiasing procedural textures is more complicated than for stored image textures. Whereas MIP maps [Williams, 1983] and summed-area tables [Crow, 1984] can be precomputed and stored for image textures, procedural textures are generated on the fly and such antialiasing techniques can not be readily applied. Supersampling is a common technique for antialiasing procedural textures but directly increases rendering time. For example, supersampling was the method used to inhibit aliasing in PixelFlow's procedural textures [Olano & Lastra, 1998]. Bandlimiting the procedural texture is also an effective technique [Norton, et al., 1982], but works easily and efficiently only on procedures based completely on spectral synthesis.

1.2. Overview

Section 2 introduces a texture model capable of synthesizing the most commonly used procedural textures (in fact all textures in Figure 1 through Figure 6) but concise enough to implement in hardware. The identification of this model allows the textures to be specified by parameters to a fixed procedure which can be simplified enough to be implemented in present-day VLSI technology.

Section 3 introduces a new method for antialiasing procedural textures based on computing a first order approximation of the color index variance over the area of a pixel. This approximation allows the antialiasing method to simulate an area sample of the textured image faster than supersampling. Unlike bandlimiting (which is a *pre*-filter), the new method is a *post*-filter that does not affect the parameters of the generation of the texture.

Section 4 exhibits the results of this model, exploring the various tradeoffs necessary to feasibly implement the model without significantly compromising image quality. An effective but reduced model can be implemented with as few as 100,000 gates, which is about 10% of the real-estate of modern consumer-level graphics processors.

2. A MODEL FOR PROCEDURAL TEXTURING

Various formalisms on procedural solid texture specifications have been proposed. Perhaps the most pervasive has been the Renderman shading language [Hanrahan & Lawson, 1990], but there are also other alternatives (e.g. [Abram & Whitted, 1990]). We propose a concise class of procedures capable of synthesizing a variety of textures and effects, but simple and direct enough to facilitate hardware implementation. The procedures are parameterized by values that completely control the type and character of the texture this model generates, such that these parameters (and the texture's color map) are the only representation of the texture that need be stored.

2.1. Analytical Model

Procedural solid texture mapping uses a mapping of the form $\mathbf{p}$: $\mathbf{R}^3 \rightarrow \mathbf{R}^4$ from solid texture coordinates $\mathbf{s} = (s,t,r)$ into a color space (R,G,B,α). (We follow the convention of using boldface to indicate vector values and functions, and italics to indicate scalar values and functions.) Some texture mapping techniques also include a homogeneous texture coordinate [Segal, *et al.,* 1992] but it remains to be explored how such a coordinate benefits procedural solid texturing. Often procedural solid textures incorporate a color map. In such cases, $\mathbf{p} = \mathbf{c} \circ f$ consisting of an implicit classification of the texture space f: $\mathbf{R}^3 \rightarrow \mathbf{R}$ and a color map $\mathbf{c}$: $\mathbf{R} \rightarrow \mathbf{R}^4$.

For a given polygon, the texture coordinate functions $\mathbf{s}(\mathbf{x}) = (s(\mathbf{x}),t(\mathbf{x}),r(\mathbf{x}))$ indicate the range of the texture coordinates with respect to screen coordinates $\mathbf{x}=(x,y)$. Hence, the procedural texture can be evaluated with respect to screen coordinates as $\mathbf{p}(\mathbf{x}) = \mathbf{c} \circ f \circ \mathbf{s}(\mathbf{x})$.

We restrict the texture map $\mathbf{p}$ to the family of functions

$$\mathbf{p}(\mathbf{s}) = \mathbf{c}\left(q(\mathbf{s}) + \sum_i a_i n(T_i(\mathbf{s})) \right), \tag{1}$$

where q: $\mathbf{R}^3 \rightarrow \mathbf{R}$ is a quadric classification function and n: $\mathbf{R}^3 \rightarrow \mathbf{R}$ is a noise function. The combination of quadrics and noise yields a specification sufficient to generate a wide variety of commonly used procedural solid textures. The affine transformations T_i control the frequency and phase of the noise functions.

2.1.1. Color Map

The color map $\mathbf{c}$ associates a color (R,G,B) with each index returned by the classification function f. The color map $\mathbf{c}$ is typically implemented as a lookup table

$$\mathbf{c}(f) = \mathbf{clut}[\text{round}(n \cdot \text{modclamp}(f))] \tag{2}$$

where **clut**[] is an array of n RGB color vectors. Color map indices returned by f are, depending on a flag parameter, either clamped to [0,1] or taken modulo one to map within the bounds of the lookup table.

2.1.2. Quadric Classification Function

The function q: $\mathbf{R}^3 \rightarrow \mathbf{R}$ in (1) is the quadric

$$q(s,t,r) = As^2 + 2Bst + 2Csr + 2Ds + Et^2 + 2Ftr + 2Gt + Hr^2 + 2Ir + J \tag{3}$$

which can more conveniently be represented homogeneously as

$$q(\mathbf{s}) = \mathbf{s}^T Q \mathbf{s} = [s,t,r,1] \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} \begin{bmatrix} s \\ t \\ r \\ 1 \end{bmatrix} \tag{4}$$

treating $\mathbf{s}$ as a homogeneous column vector [Blinn, 1982].

The quadric function supports the spherical, cylindrical, hyperbolic and parabolic classification of space for texturing.

2.1.3. Noise Function

The function n: $\mathbf{R}^3 \rightarrow \mathbf{R}$ in (1) is an implementation of the Perlin noise function [Perlin, 1985]. The values a_i control the amplitude of the noise function, whereas the affine transformation T_i controls the frequency and phase of each noise component. There are a fixed number of noise components available, and this limit is typically between four and eight in typical texturing examples.

2.2. Texture Examples

The space of solid textures spanned by (1) covers the textures most commonly found in procedural solid texturing. The four fundamental procedural solid textures are: wood, clouds, marble and fire.

2.2.1. Wood

The texture model generated the wood texture shown in Figure 1, by using the quadratic function to classify the texture space into a collection of concentric cylinders [Peachey, 1985]. Waviness in the grain is created by modulation of a noise function

$$f(s,t,r) = s^2 + t^2 + n(4s,4t,r). \tag{5}$$

The color map consists of a modulo-one linear interpolation of a light "earlywood" grain and a darker "latewood" grain. The quadric classification makes the early rings wider than the later rings, which is to a first approximation consistent with tree development.

2.2.2. Clouds

Cloudy skies are made with a fractal $1/f$ sum of noise

$$f(\mathbf{s}) = \sum_{i=1}^{4} 2^{-i} n(2^i \mathbf{s}). \tag{6}$$

The texture described by (6) is mapped onto a very large high-altitude polygon parallel to the ground plane in Figure 3, resulting in clouds that become more dense in the distance due to perspective-corrected texturing coordinate interpolation. The color map is a clamped linear interpolation from blue to white. The water is the same procedural texture with a blue-to-black colormap.

2.2.3. Marble

Marble uses the noise function to distort a linear ramp function of one coordinate [Perlin, 1985]

$$f(s,t,r) = r + \sum_{i=1}^{4} 2^{-i} n(2^i s, 2^i t, 2^i r)). \tag{7}$$

The color map consists of a modulo-one table of colors from a cross section of the marble. Figure 2 demonstrates the marble texture on a cube, and the solid texturing again aligns the texture details on the edges of the cube. Continuously increasing the noise amplitude animates the formation of the ripples in the marble, simulating the pressure and heating process involved in the development of marble [Ebert, 1994].

2.2.4. Fire

Like marble, fire is simulated by offsetting a texture coordinate with fractal noise [Musgrave & Mandelbrot, 1989]. The fire example shown in Figure 4 was textured onto a single polygon and modeled as

$$f(s,t,r) = r + \sum_{i=1}^{4} 2^{-i} n(2^i s, 0, 2^i r + \phi)). \quad (8)$$

Continuously varying the noise phase term ϕ animates the fire texture.

2.2.5. Planet

A wide variety of different worlds, such as the one shown in Figure 5, can be generated by applying fractal textures, such as (6), to spheres. The color map for such images resembles a cartographic "legend." The cloudy atmosphere was rendered on the same sphere "over" the planet in a second pass using a color map with varying opacity values.

2.2.6. Moonrise

The moonrise in Figure 6 was rendered completely using synthesized textures, without any other kind of shading. The moon is a sphere with a fractal texture. The clouds were rendered on a single polygon perpendicular to the viewer and imposed over the moon. The water was rendered with a single polygon extending off to infinity. The highlight on the water was faked with two triangles textured using (7) with a partially transparent color map.

3. ANTIALIASING

Image texture aliases occur due to texture magnification and minification. Texture magnification occurs when the texture image itself contains too few samples such that a single texture element projects to several screen pixels. Texture minification results when the projection of the texture image covers too few pixels and several texture elements project to the same screen pixel. Modern texture mapping hardware inhibits aliases due to texture magnification by bilinear or bicubic interpolation of the appropriate texture elements. Such hardware inhibits texture minification aliases through the use of a MIP map that precomputes lower resolution versions of the texture, and samples the MIP map using trilinear or tricubic interpolation of neighboring pixels at the appropriate resolution level.

Aliases of synthesized textures do not fall into such categories since there is no fixed image resolution. Each such texture will exhibit some form of aliasing if sampled below twice the highest frequency in the texture's spectrum, which may be infinite for some textures. Hence, procedural textures do not suffer from magnification aliases, but require filtering to remove frequencies above the Nyquist limit to avoid minification aliases.

Synthetic textures could be antialiased by precomputing them, storing the results in MIP-mapped image textures. However, such

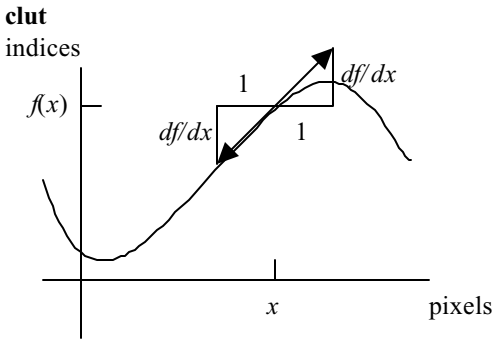


Figure 7: The derivative *df/dx* approximates the extent of the color map indices one pixel in either direction. Half of the derivative estimates the variation in color map indices half of a pixel in either direction.

an antialiasing technique would remove the flexibility such textures provided, and would also consume a tremendous amount of space when used on solid textures. Band limiting the output of the texture map removes aliases by prefiltering the texture before sampling [Norton, et al., 1982], but is difficult to implement in a generalized texturing environment. Supersampling the texture degrades time perfomance and arbitrarily increases the complexity of the hardware implementation.

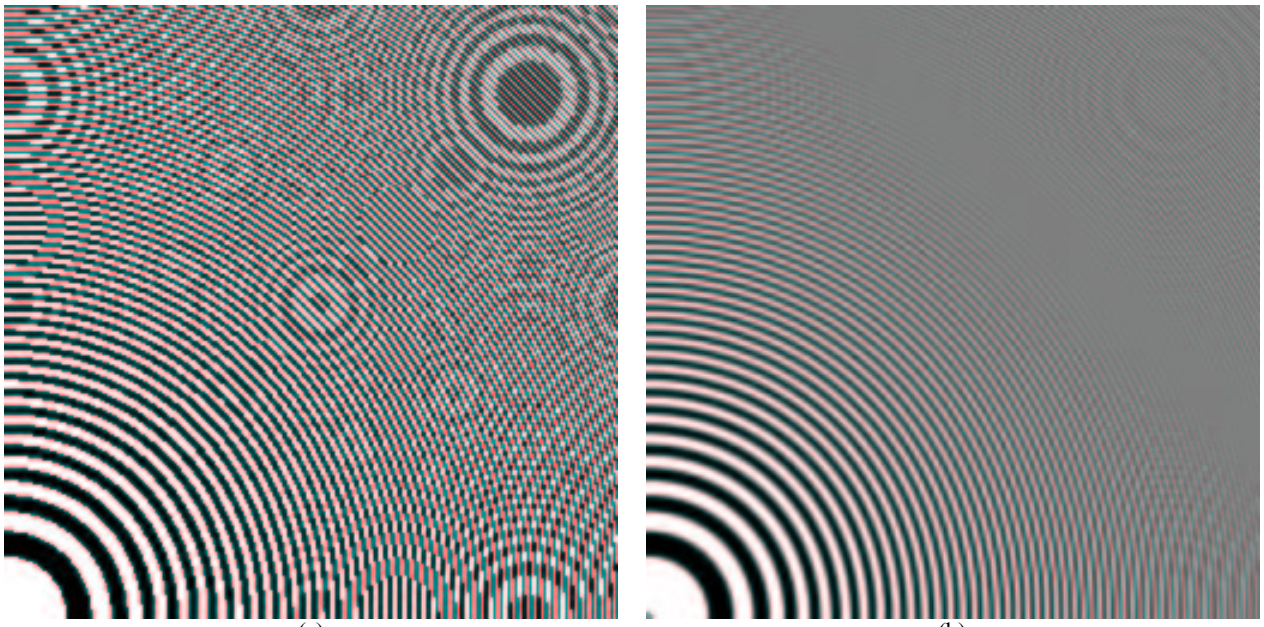
Instead, we analyze the function $\mathbf{p}(\mathbf{x})$ that textures pixels to determine the width of a box filter that would eliminate the aliasing frequencies from the spectrum of the synthesized texture. Several have described techniques for antialiasing procedural textures by antialiasing the textures' colormaps [Rhoades, *et al.*, 1992], [Worley, 1994]. In the next section, we provide a more rigorous mathematical justification and derivation of the technique, resulting in an ideal filter width for the texture which is used to box filter to the procedural texture by averaging the elements of the color table that the texture procedure generates over the support of the filter.

3.1. Texture Filtering via Color Table Filtering

Consider a domain D on the screen consisting of pixels whose color is determined solely by the projection of a single procedurally texture mapped polygon. We assume the color map indices generated by the procedural texture are continuous across the polygon. Let $a = \min_D f(\mathbf{x})$ be the least possible color map index used in the pixels in D, and let $b = \max_D f(\mathbf{x})$ be the greatest such index. Then we assume

$$\frac{\int_D \mathbf{p}(\mathbf{x})d\mathbf{x}}{\int_D d\mathbf{x}} \approx \frac{\int_a^b \mathbf{c}(f)df}{b-a} \quad (9)$$

the average color in D is sufficiently approximated by the average of the color table entries between indices a and b. As shown in Figure 7, we provide a first-order approximation of the bounds a and b used in the RHS of (9) by differentiating the texture function $f(\mathbf{x})$ and setting $a = f(\mathbf{x}) - \|\nabla f(\mathbf{x})\|/2$ and $b = f(\mathbf{x}) + \|\nabla f(\mathbf{x})\|/2$. If either a or b or both fall outside the bounds of the color table, then the boundary of the color table is extended using



(a) (b)

Figure 8: Zone plate aliased (a) and filtered (b).

either the modulo or clamp operators according to the modclamp flag.

The remainder of this section describes this differentiation in detail, applies efficient methods for integrating the color map to determine the numerator of the RHS of (9), and demonstrates the results.

3.2. Differentiating the Texture Procedure

The magnitude of the gradient $\nabla f = (\partial f/\partial x, \partial f/\partial y)$ indicates the width of the appropriate filter on the color map. From (1), we have that the gradient of f is

$$\nabla f = \nabla q + \sum_i a_i \nabla n_i \tag{10}$$

where n_i is the ith noise function: $n(T_i(\mathbf{s}))$. From (3) we have that the gradient of q is

$$\begin{aligned} \nabla q(\mathbf{x}) &= \mathbf{s}^T Q \frac{d\mathbf{s}}{d\mathbf{x}} + \left(\frac{d\mathbf{s}}{d\mathbf{x}}\right)^T Q\mathbf{s} \\ &= 2\mathbf{s}^T Q \frac{d\mathbf{s}}{d\mathbf{x}} \\ &= 2\begin{bmatrix} s & t & r & 0 \end{bmatrix} \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix} \begin{bmatrix} \frac{\partial s}{\partial x} & \frac{\partial s}{\partial y} \\ \frac{\partial t}{\partial x} & \frac{\partial t}{\partial y} \\ \frac{\partial r}{\partial x} & \frac{\partial r}{\partial x} \\ 0 & 0 \end{bmatrix} \end{aligned} \tag{11}$$

since Q is symmetric.

The derivative of the noise terms are given by

$$a_i \nabla n(T_i\mathbf{s}(\mathbf{x})) = a_i \frac{dn(T_i\mathbf{s}(\mathbf{x}))}{d\mathbf{s}} T_i \frac{d\mathbf{s}}{d\mathbf{x}}. \tag{12}$$

The gradient $dn/d\mathbf{s} = [\partial n/\partial s \ \partial n/\partial t \ \partial n/\partial r \ 0]$ is also known as the function DNoise [Perlin, 1985].

The value $d\mathbf{s}/d\mathbf{x}$ is the Jacobian of the texture coordinates $\mathbf{s}$ with respect to the screen coordinates $\mathbf{x}$. The values of $d\mathbf{s}/dx$ is computed during the scan conversion of the polygon as the perspective-corrected pixel increments. The values of $d\mathbf{s}/dy$ can be computed for each triangle using the plane equation and performing a perspective-correcting division.

3.3. Filtering the Color Table

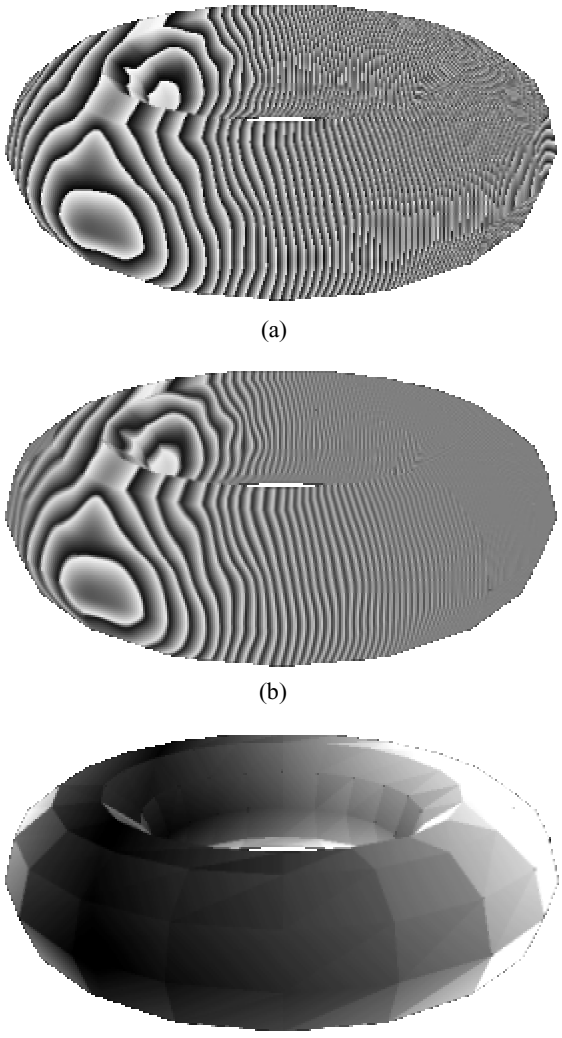
The filtering of color map values can be evaluated efficiently using either a color table MIP map or a summed area color table.

3.3.1. Color table MIP map

MIP maps are commonly used in standard texturing systems to prefilter image textures and sample from the prefiltered texture when the texture is minified (insufficiently sampled by the image pixels) [Williams, 1983].

One may also create a MIP map of a color table. The process begins with the n-element full resolution color table $\mathbf{clut}_1[]$. Then neighboring colors in the table are averaged to create a half-resolution $n/2$-element color table $\mathbf{clut}_2[]$. This process is repeated until a one-element color table $\mathbf{clut}_{\lg n}[]$ results, representing the average color of the entire color table.

Given a filter width w, let i = floor(lg w). Then the proper resolution color table from the mip map is selected and the color indexed is returned as $\text{clut}_i[f/i]$ (or more accurately the linear or cubic interpolation of the values of $\text{clut}_i[f/i]$ and $\text{clut}_{i+1}[f/(i+1)]$).



(a) (b) (c)

Figure 9: Torus rendered with wood texture (a) is antialiased (b) using filterwidths shown in (c) ranging from one (black) to 256 (white).

3.3.2. Summed area color table

Image textures are also antialiased efficiently using the summed area table [Crow, 1984]. A summed area table transforms information into a structure that can quickly perform integration, specifically a box filtering operation.

The summed area color table consists of a table where each entry consists of the sum of all elements in the color table including the current entry's element

$$\mathbf{csat}[i] = \sum_{j=0}^{i} \mathbf{clut}[j] \tag{13}$$

or recurrently as $\mathbf{csat}[i] = \mathbf{csat}[i\text{-}1] + \mathbf{clut}[i]$. The current entry's element can be recovered by subtracting the previous summed area element from the current summed area element as

$$\mathbf{clut}[i] = \mathbf{csat}[i] - \mathbf{csat}[i\text{-}1] \tag{14}$$

for i > 0. Box filtering the color map entries for a given filter width is computed as

$$(\mathbf{csat}[f + w/2] - \mathbf{csat}[f - w/2])/w. \tag{15}$$

Special care must be taken for the cases where the support of the filter crosses the bounds of the color table. For the following cases let N is the number of entries in the color table.

- $w \geq N$: Return the average of the entire color map: $\mathbf{csat}[\text{N-1}]/N$.
- $f + w/2 \geq N$:
  mod: $(\mathbf{csat}[f + w/2 - \text{N}] + \mathbf{csat}[\text{N-1}] - \mathbf{csat}[f - w/2 - 1])/w$.
  clamp: $((f+w/2-(\text{N-1}))\mathbf{clut}[\text{N-1}] + \mathbf{csat}[\text{N-1}] - \mathbf{csat}[f-w/2\text{-}1])/w$.
- $f - w/2 < 0$:
  mod: $(\mathbf{csat}[f + w/2] + \mathbf{csat}[N\text{-}1] - \mathbf{csat}[N + f - w/2 - 1])/w$.
  clamp: $(-(f - w/2)\ \mathbf{clut}[0] + \mathbf{csat}[f + w/2])/w$.

An alternative to performing the above computations at render time is to use the above formulae to precompute a color summed area table three times as long, ranging from $-N$ to $2N - 1$.

3.4. Examples

The derivations in Section 3.2 show that procedural textures produce aliasing artifacts from three possible places.

1. **Quadric Variation:** The quadric classification changes too quickly: $\|dq/d\mathbf{s}\|$ too large.
2. **Noise Variation:** The noise changes too quickly: $a_i\|dn(T_i\mathbf{s})/d\mathbf{s}\|$ too large.
3. **Texture Coordinate:** The texture coordinates change too quickly: $\|d\mathbf{s}/d\mathbf{x}\|$ too large.

Each of these components can create a signal containing frequencies exceeding the Nyqist limit of the pixel sampling rate.

Figure 8 demonstrates quadratic variation aliasing (type #1) with a zone plate constructed from the procedure

$$f(s,t,r) = 50s^2 + 50t^2 \cdot \tag{16}$$

rendered with an extremely harsh "zebra" color map. Analysis of (16) shows that the aliases are governed by $\nabla f = dq/d\mathbf{s}\ d\mathbf{s}/d\mathbf{x}$, with $dq/d\mathbf{s} = (100\ s, 100\ t)$. The zone plate was plotted at a resolution of 256^2 and over the unit square in texture coordinate space, hence $\partial s/\partial x = \partial t/\partial y = 1/256$. Setting the colormap filter width to $(100\ s + 100\ t)/256$ reduces the aliases to the point of being barely noticable.

Noise variation aliases (type #2) happen in concert with texture coordinate aliasing (type #3), since in a single scene the frequency and amplitude of noise is constant, and only varies across the image with distance from the viewer. For example, the clouds on the horizon in Figure 3 do not alias near the horizon because the filter width is scaled in part by the noise function derivative, and increases as the magnitude of $d\mathbf{s}/d\mathbf{x}$ increases. In the distance as the projection of the noise reaches the Nyquist limit, the filter width reaches the size of the entire color table, yielding a homogeneous hazy blue color.

Figure 9 illustrates all three types of texture aliasing on a torus. The centerline of the woodgrain rings passes through the left side of the torus, creating grain of increasing frequency on the right. Hence the filterwidth increases from the left to the right side of

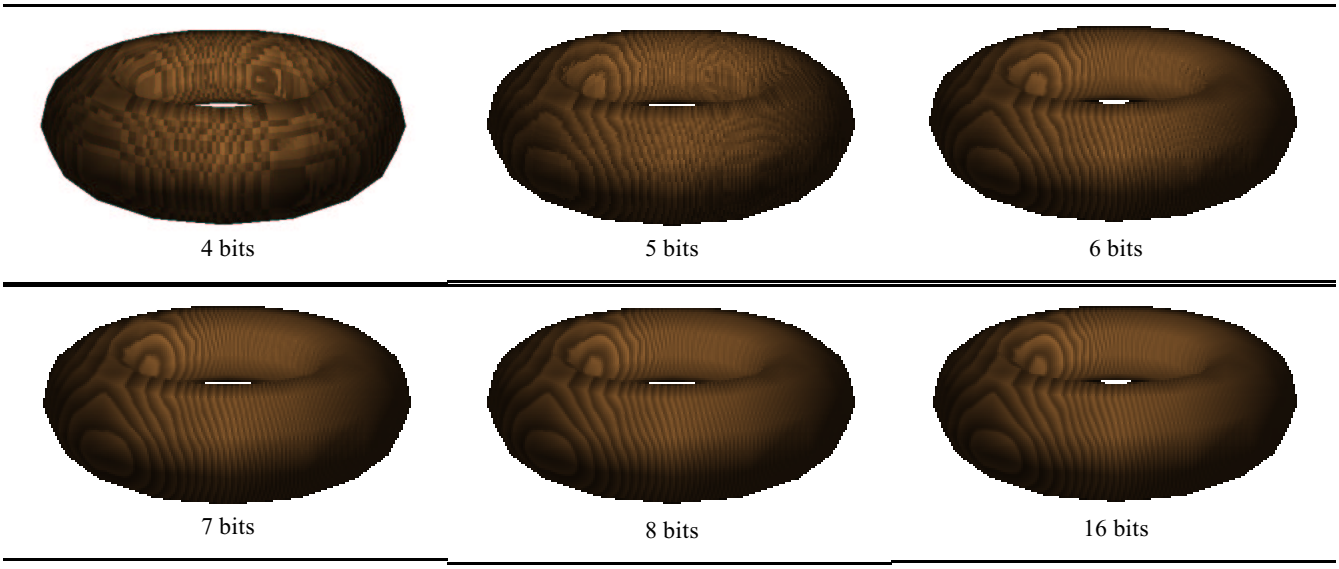


Figure 10: The effect of numerical precision on texture appearance.

the torus demonstrating quadric variation (type #1) aliasing. The amplitude and frequency of the noise term remains constant over the torus object, and so causes a uniform increase of the filterwidth due to noise variation (type #2) aliasing. The polygons on the silhouette of the torus have larger filterwidths than their neighbors, demonstrating texture coordinate (type #3) aliasing.

4. Results

The goal of the previous sections was to simplify the synthesis of antialiased solid textures. In this section, we describe and demonstrate software and simulated hardware implementations, and document some of the tests performed in the process.

4.1. Software Implementation

The basic tool of this research is a simulator that implements in fixed point arithmetic the texture synthesis model along with its associated filtering and color table mechanisms, as well as a prototype rasterizer. This simulator is responsible for all of the textured images in this paper. While the textures themselves were antialiased, the polygon edges were not. In fact, we avoided the temptation to use many small polygons to create smoother surfaces and silhouettes in order to better demonstrate the ability of procedural textures instead of geometry to provide visual detail.

This simulator serves as an antialiasing procedural texturing shader, and could be incorporated as a plug-in to existing software rendering systems. This simulator also serves as the basis of an extension to OpenGL, which already supports solid texture coordinates. The current implementation uses the OpenGL feedback buffer to collect the transformed polygons in screen coordinates for rasterization by the simulator [Carr & Hart, 1999]. The resulting textured raster image generated by the simulator is then combined with the raster image generated by OpenGL's rasterization engine using the associated z-buffers to negotiate visibility. Hence the simulator integrates synthesized solid textures into OpenGL's existing texturing, lighting and modeling system.

4.2. Hardware Implementation

A complete implementation of the model can be realized in VLSI with 1.25 million gates, resulting in the image quality shown in Figure 1 through Figure 6. A reduced and approximated version of the texture synthesis model can be implemented in as few as 100,000 gates. Sample images from such an implementation are exhibited in Figure 11.

Overall, the compromises in image quality necessary to implement the model in 100,000 gates appear minor, and the effects are very subtle. Some texture coordinate aliasing is noticable on the polygons of the teapots closest to the viewer. The character of the water, sky, planet and moonrise are slightly smoother due to a reduction in the number of noise function evaluations. The teapots and fire have noticable artifacts due to a linear approximation to the noise function.

4.3. Precision Tests

Several tests have been conducted to determine the texture coordinate precision necessary to avoid magnification aliases [Kameya & Hart, 1999]. Figure 10 shows the results of tests with a 512^2-pixel scene of a coarsely-triangulated objects computed using a variety of texture coordinate precisions.

4.4. Animation Tests

The seascape was animated to determine the effectiveness of the antialiasing technique. The seascape scene (Figure 3) was the most taxing on the colormap filtering algorithm because it textures infinite planes. Two animations of flights into the horizon were generated, one with and one without filtering. The unfiltered animation resulted in severe aliasing in the form of distracting noise near the horizon. The filtered animation significantly reduced these aliases, although some very slight flicker is still observable. This subtle flicker seems to be an inevitable compromise of the colormap-averaging filter in that removing the flicker results in textured planes that get too blurry too soon before reaching the horizon.

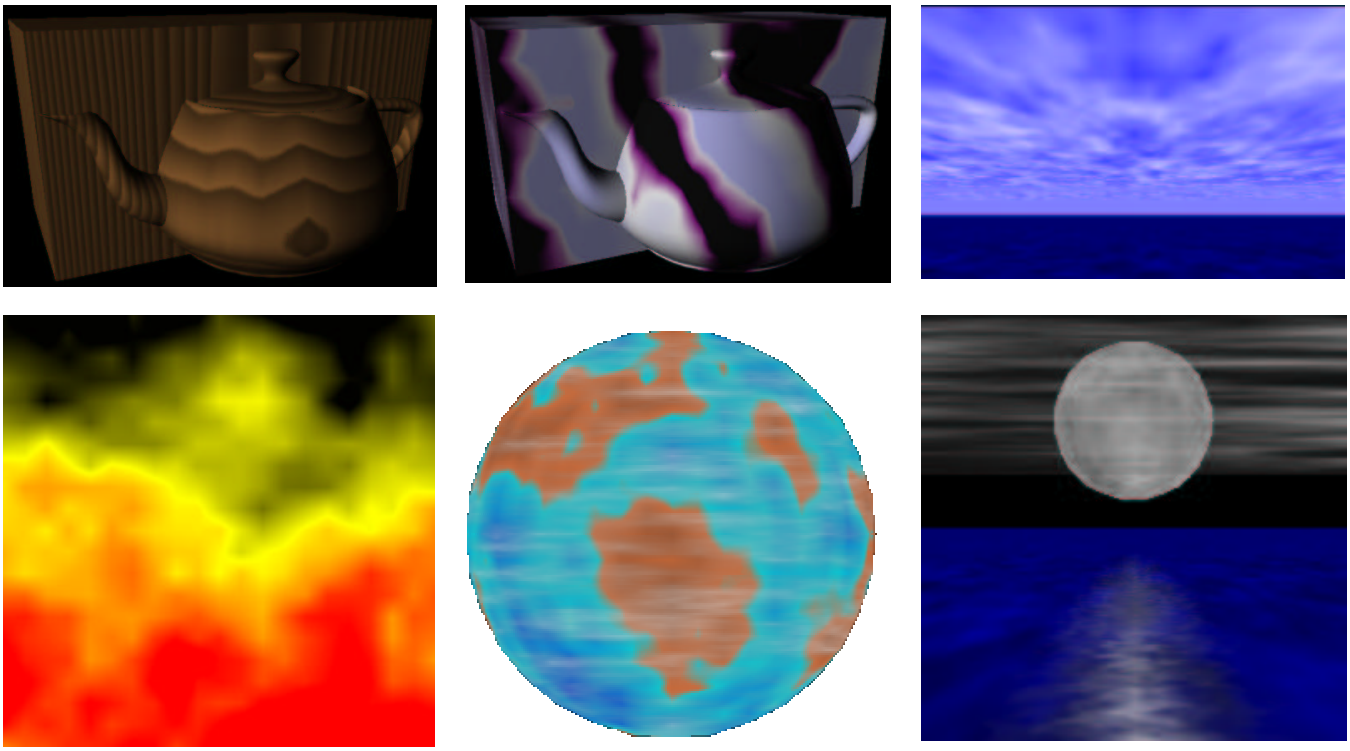


Figure 11: 100,000 gate simulations of Figure 1 through Figure 6.

The flame shown in Figure 11 was also animated to determine how effectively they would appear in the hardware implementation. The rectilinear grid basis of the noise functions is clearly evident due to the reduced number of noise octaves and the tri-linear interpolation. However the animation does clearly resemble burning flames and would sufficently represent such in typical consumer real-time graphics applications.

5. Conclusion

We set out to formalize a model for synthesizing popular procedural solid textures, and analyzed this model to derive an effective antialiasing scheme and an efficient hardware implementation. We showed that the model is capable of simulating the common procedural textures of wood, clouds, marble and fire, but is also simple enough to adequately implement in hardware.

Often textures are animated, to simulate fire, billowing clouds and other dynamic effects. Animation of texture map images requires a significant amount of texture memory and fast CPU access to the texture memory. The procedural texturing hardware will be capable of real-time animation of clouds billowing, fire burning and marble forming.

PixelFlow defered shading until after all of the rasterization was completed [Molnar, *et al.,* 1992]. It stored all of the shading information in the frame buffer, such that each pixel was shaded only once regardless of the number of polygons that overlapped it. The procedural texturing hardware described in this paper could be used to texture such pixels if the texture index, coordinates and Jacobian were stored in the framebuffer.

5.1. Future Work

This work only scratches the surface of procedural texturing hardware. Procedural texturing inexpensively overcomes the fundamental graphics texture rendering problems of memory bandwidth. With the success of this particular model, we expect other more sophisticated texturing models will be developed. The connotation of procedural texturing is that an actual program is run to generate the texture. While our model uses a fixed program with parameters controlling the character of its output, future procedural texturing hardware might be designed to permit uploading of texture programs. While such machines already exist (e.g. the Pixel Machine, Pixel Planes) there is no restriction on the texturing programs. Hence the user is burdened responsibility of antialiasing. Restricting the language used to write a procedural shader can increase the quality of its output, as it allows the hardware to better analyse the program to predict the aliases its output may contain, and automatically take measures to inhibit those aliases.

The antialiasing technique was derived from the model, but there is nothing specific to the model that makes this antialiasing technique work. Hence the color map antialiasing technique could be generalized and applied to any procedural texture so long as the derivatives are available. Computation of these derivatives is straightforward for this simple model, but could be quite complicated for true procedural textures described in a programming language. The error associated with approximation (9) should also be investigated further.

The colormap of the planet in Figure 5 is not continuous, jumping from a sandy color to an aquamarine to mark the coastlines of the world. As the filterwidth increases due to the noise contributions, this sharp coastline diffuses into a muddy color inbetween. A

more sophisticated antialiasing system might mark such jump discontinuities in the colormap and affect the filterwidth in these areas to further inhibit this artifact.

The noise function used was adapted from Rayshade [Skinner & Kolb, 1991], which uses cubic blending functions on a lattice of random numbers. This particular version lends itself to efficient hardware implementation, but the details of such an implementation are left as future work.

Procedural hardware need not be limited to just texture. Procedural hardware bump mapping, displacement mapping and shading in general seem to be logical extensions of this work. Recently, minor extensions to existing graphics pipelines for increased shading language support have been proposed [McCool & Heidrich, 1999]. Further extension might lead to the generation of procedural geometry that would overcome the bandwidth problem of transmitting polygons from the host to the graphics processor.

5.2. Acknowledgments

This research is supported in part by Evans and Sutherland Computer Corp., with a matching grant by the Washington Technology Center. This research was performed in part using the facilities of the Image Research Laboratory in the School of EECS at Washington State University.

Bibliography

[Abram & Whitted, 1990] Abram, Gregory D. and Turner Whitted. Building block shaders. *Computer Graphics 24*(4), (Proc. SIGGRAPH 90), Aug. 1990, pp. 283-288.

[Akeley, 1993] Akeley, Kurt. Reality engine graphics. *Computer Graphics 27*, Annual Conference Series, (Proc. SIGGRAPH 93), July 1993, pp. 109-116.

[Beers, *et al.,* 1996] Beers, Andrew C., Maneesh Agrawala and Navin Chaddha. Rendering from Compressed Textures. *Computer Graphics*, Annual Conference Series, (Proc. SIGGRAPH 96), Aug. 1996, pp. 373-378.

[Blinn, 1982] Blinn, James F., A generalization of algebraic surface drawing *ACM Transactions on Graphics 1*(3), July 1982, pp. 235-256.

[Carr & Hart, 1999] Carr, Nate and John C. Hart. APST Antialiased Procedural Texturing Interface for OpenGL. Proc. Western Computer Graphics Symposium. March 1999, pp. 46-55.

[Crow, 1984] Crow, Franklin C. Summed area tables for texture mapping. *Computer Graphics 18*(3), (Proc. SIGGRAPH 84), July 1984, pp. 137-145.

[Ebert, 1994] Ebert, David. Animating Solid Spaces: Animating Solid Textures. Chapter in: Texturing and Modeling: A Procedural Approach, Ebert, D., Ed. Academic Press Professional, Boston, 1984, pp. 165-170.

[Hanrahan & Lawson, 1990] Hanrahan, P. and J. Lawson. A language for shading and lighting calculations. *Computer Graphics 24*(4), (Proc. SIGGRAPH 90), Aug. 1990, pp. 289-298.

[Kameya & Hart, 1999] Kameya, Masaki and John C. Hart. Bit width necessary for solid texturing hardware. Proc. Western Computer Graphics Symposium. March 1999, pp. 121-126.

[Molnar, *et al.,* 1992] Molnar, Steven, John Eyles and John Poulton. PixelFlow: High-speed rendering using image composition. *Computer Graphics 26*(2), (Proc. SIGGRAPH 92), July 1992, pp. 231-240.

[Montrym, *et al.,* 1997] Montrym, John S., Daniel R. Baum, David L. Dignam and Christopher J. Migdal. InfiniteReality: A real-time graphics system. *Computer Graphics,* Annual Conference Proceedings, (Proc. SIGGRAPH 97), Aug. 1997, pp. 293-302.

[Musgrave & Mandelbrot, 1989] Musgrave, F. Kenton and Benoit B. Mandelbrot. Natura Ex Machina. *IEEE Computer Graphics and Applications 9*(1), Jan. 1989, p. 4-7.

[McCool & Heidrich, 1999] McCool, Michael D. and Wolfgang Heidrich. Texture Shaders. Proc. Eurographics-SIGGRAPH Graphics Hardware Workshop, Aug. 1999.

[Norton, et al., 1982] Norton, Alan, Alyn P. Rockwood and Phillip T. Skolmoski. Clamping: A method for antialiased textured surfaces by bandwidth limiting in object space. *Computer Graphics 16*(3), (Proc. SIGGRAPH 82), July 1982, pp. 1-8.

[Olano & Lastra, 1998] Marc Olano and Anselmo Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System. Computer Graphics, Annual Conference Proceedings, (Proc. SIGGRAPH 98), July 1998, pp. 159-168.

[Peachey, 1985] Peachey, Darwyn. R. Solid texturing of complex surfaces. *Computer Graphics 19*(3), (Proc. SIGGRAPH 85), July 1985, pp. 279-286.

[Perlin, 1985] Perlin, Ken. An image synthesizer. *Computer Graphics 19*(3), (Proc. SIGGRAPH 85), July 1985, pp. 287-296.

[Potmesil & Hoffert, 1989] Potmesil, Michael and Eric M. Hoffert. The Pixel Machine: A parallel image computer. *Computer Graphics 23*(3), (Proc. SIGGRAPH 89), July 1989, pp. 69-78.

[Rhoades, *et al.*, 1992] Rhoades, John, Greg Turk, Andrew Bellm Andrei State, Ulrich Neumann and Amitabh Varshney. Real-Time Procedural Textures. Proc. Interactive 3-D Graphics Workshop, 1992. pp. 95-100.

[Segal, *et al.,* 1992] Segal, Mark, Carl Korobkin, Rolf van Widenfelt, Jim Foran and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics 26*(2), (Proc. SIGGRAPH 92), July 1992, pp. 249-252.

[Skinner & Kolb, 1991] Skinner, Robert and Craig E. Kolb. noise.c (file in the Rayshade raytracing system).

[Torborg & Kajiya, 1996] Torborg, Jay and James T. Kajiya. Talisman: Commodity realtime 3D graphics for the PC. *Computer Graphics* Annual Conference Proceedings, (Proc. SIGGRAPH 96), Aug. 1996, pp.353-363.

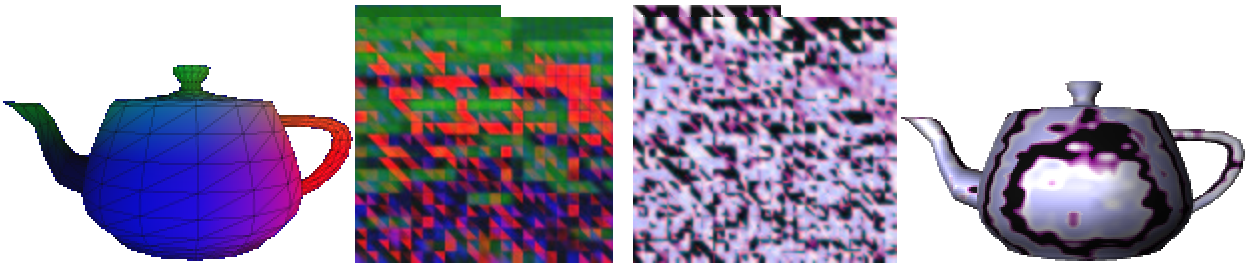
[Williams, 1983] Williams, Lance. Pyramidal parametrics. *Computer Graphics 17*(3), (Proc. SIGGRAPH 83), July 1983, pp. 1-11.

[Worley, 1994] Steven Worley. Practical Methods for Texture Design: Antialiasing. Chapter in: Texturing and Modeling: A Procedural Approach, Ebert, D., Ed. Academic Press Professional, Boston, 1984, pp. 117-124.

Real-Time Procedural Solid Texturing

Nathan A. Carr John C. Hart

Department of Computer Science
University of Illinois, Urbana-Champaign



(a) (b) (c) (d)

Figure 1. Solid texture coordinates stored as vertex colors of a model (a) are rasterized into a texture atlas (b). A procedural shader replaces the interpolated solid texture coordinates with colors (c), which are applied to the object using texture mapping.

Abstract

Shortly after its introduction in 1985, procedural solid texturing became a must-have tool in the production-quality graphics of the motion-picture industry. Now, over fifteen years later, we are finally able to provide this feature for the real-time consumer graphics used in videogames and virtual environments. A texture atlas is used to create a 2-D texture map of the 3-D solid texture coordinates for a given surface. Applying the procedural texture to this atlas results in a view-independent procedural solid texturing of the object.

Texture atlases are known to suffer from sampling problems and seam artifacts. We discovered that the quality of this texturing method is independent of the continuity and distortion of the atlas, which have been focal points of previous atlas techniques. We instead develop new meshed atlases that ignore continuity and distortion in favor of a balanced distribution of as many texture samples as possible. These atlases are seam-free due to careful attention to their rasterization in the texture map, and can be MIP-mapped using a balanced mesh-clustering algorithm.

Techniques for fast procedural synthesis are also investigated, using either the host processor or with multipass graphics processor operations on the texture map. We used these atlas and synthesis techniques to create a real-time procedural solid texture design system.

CR Categories: I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism (color, shading and texture).

Keywords: Atlas, mesh partitioning, MIP-map, multipass rendering, procedural texturing, solid texturing, texture mapping.

1. Introduction

The concept of procedural solid texturing is well known [32][37], and has found widespread use in graphics [6]. Solid texturing simulates a sculpted appearance and directly generates texture coordinates regardless of surface topology. Procedural texturing makes solid texturing practical by computing the texture on demand (instead of accessing a stored volumetric array), and at a level detail limited only by numerical precision. These features were quickly adopted for production-quality rendering by the entertainment industry, and became a core component of the Renderman Shading Language [11].

With the acceleration of graphics processors outpacing the exponential growth of general processors, there have been several recent calls for real-time implementations of procedural shaders, e.g. [12][38]. Real-time procedural shaders would make videogame graphics richer, virtual environments more realistic and modeling software more faithful to its final result. Section 2 describes previous implementations of real-time procedural texturing and shading systems, all requiring special-purpose graphics supercomputers or processors.

Peercy *et al.* [35] recently took a large step toward this goal by developing a compiler that translated Renderman shaders into multipass OpenGL code. While complex Renderman shaders could not yet be rendered in real-time, this compiler showed that their implementation on graphics accelerators was at least feasible. They created new interactive shading language, ISL, to produce more efficient OpenGL shaders.

Unfortunately, ISL did not introduce any new techniques for solid texturing, supporting it instead with texture volumes. While modern graphics accelerator boards now have enough texture memory to store a moderate resolution volume, and some even support texture compression, storing a 3-D dataset to produce a 2-D surface texture is inefficient and an unnecessarily wasteful use of texture memory. Applying procedural texturing operations to an entire texture volume also wastes processing time.

Apodaca [1] described how the texture map can be used to store the shading of a model. His technique shaded a mesh in world coordinates, but stored the resulting colors in a second "reference" copy of the mesh embedded in a 2-D texture map. The mesh could then be later shaded by applying the texture map instead of computing its original shading.

We can use this technique to support view-independent procedural solid texturing. Consider a single triangle with 3-D solid texture

Authors' address: Urbana, IL 61801. {nacarr, jch}@uiuc.edu.

coordinates[1] $\mathbf{s}_i$ and 2-D surface coordinates $\mathbf{u}_i$ assigned to its vertices $\mathbf{x}_i$ for $i = 1,2,3$. Figure 1a shows such triangles, plotted in model coordinates with color indicating their solid coordinates. We apply a procedural solid texture to the triangle $(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3)$ in three steps. The first step rasterizes the triangle into a texture map using its surface texture coordinates $(\mathbf{u}_1,\mathbf{u}_2,\mathbf{u}_3)$. This rasterization interpolates its vertices' solid texture coordinates $\mathbf{s}_i$ across its face. Figure 1b shows each pixel (u,v) in the rasterization now contains the interpolated solid texture coordinates $\mathbf{s}(u,v)$. The second step executes a texturing procedure $\mathbf{p}()$ on these solid texture coordinates, resulting in the color $\mathbf{c}(u,v) = \mathbf{p}(\mathbf{s}(u,v))$ shown in Figure 1c. This color table $\mathbf{c}(u,v)$ is a texture map that we apply to the original triangle $(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3)$ via its surface coordinates $\mathbf{u}_i$, resulting in the view-independent procedural solid texturing shown in Figure 1d.

This atlas technique was implemented as a tool to preview procedural solid textures in recent modeling packages [2], [45] though it suffered from sampling problems. Lapped textures [40] also used a texture atlas to allow the lapped texture swatches to be applied in a simple texture mapping operation, noting "the atlas representation is more portable, but may have sampling problems."

Section 3 describes the texture atlas in detail, and analyzes the artifacts it can cause. Poor coverage of the texture map by the atlas causes aliasing, whereas discontinuities in the atlas cause seams in the textured surface. Section 4 describes new atlases that overcome these artifacts, with atlases that cover more of the texture map and distributing the resulting samples more evenly to reduce texture magnification aliases. Section 4.3 describes how an atlas that can be MIP mapped to eliminate texture minification aliases.

The use of an atlas enables procedural texturing operations to be applied to the texture map, and Section 5 describes how this step can be implemented efficiently on both the host and the graphics controller. Section 6 concludes with an interactive procedural solid texture editor, other applications of these methods and ideas for further investigation.

2. Previous Work

There have been several implementations of real-time procedural solid texturing over the past fifteen years, though they have either required high-performance graphics computers or special-purpose graphics hardware.

Procedural solid texture has been available on parallel graphics supercomputers, such as the AT&T Pixel Machine [39] and UNC's Pixel Planes 5 and PixelFlow [26]. The Pixel Machine in fact was used as a platform for exploring volumetric procedural solid texture spaces [36].

Rhoades *et al.* [42] developed a specialized assembly language, called T-code, for procedural shading on Pixel Planes 5. The T-code interpreter included automatic differentiation to estimate the variation of the procedure across the domain of a pixel. This estimate of the variation was used as a filter width to antialias the procedural texture, by averaging the range of colors the procedure could generate within the pixel.

Olano *et al.* [30] implemented a real-time subset of the Renderman shading language on Pixel Flow, including the ability to synthesize procedural solid textures. Standard Renderman shader tools including automatic differentiation and clamping [28] were used to antialias the procedural textures.

Hart *et al.* [14] designed a VLSI processor based around a single function capable of generating several of the most popular procedural solid textures. Procedural solid textures were transmitted to this hardware as a set of parameters to the texturing function. The derivative of the function was also implemented to automatically antialias the output, à la [42].

Current graphics libraries such as OpenGL [44] and Direct3D [24] support solid texturing with the management of homogeneous 3-D texture coordinates, and recent versions of these libraries support three-dimensional texture volumes that can be MIP-mapped to support antialiasing.

Peercy et al. [35] developed a compiler that translated the Renderman shading language into OpenGL source code. The technique used multi-pass rendering and requires an OpenGL 1.2 implementation with its imaging subset, as well as the floating-point-framebuffer and pixel-feedback extensions. As mentioned in the introduction this method depends on texture volumes for solid texturing.

3. The Texture Atlas

A (*surface*) *texture mapping* $\mathbf{u} = \phi(\mathbf{x})$ is a function from a surface into a compact subset of the plane called the *texture map*. The texture mapping need not be continuous, but usually consists of piecewise continuous parts $\phi_i()$ called *charts*. The area on the surface in model coordinates is called the *chart domain* whereas the area the domain maps to in the texture map is called the *chart image*. The collection of charts that forms a texture mapping $\phi() = \cup \phi_i()$ is called an *atlas* [27]. If the surface texture mapping is one-to-one, then its inverse $\phi^{-1}()$ is a *parameterization* of the surface. Atlases often (but not always) parameterize the surface, such that each pixel in the texture map represents a unique location on the object surface[2].

Hence parameterization methods could be used to generate atlases. For example, MAPS [19] parameterizes a mesh of arbitrary topological type, using a simplified version of the mesh embedded in three-space to serve as the base domain of smoothed piecewise barycentric parameterizations. This base mesh and the parameterization it supports could be flattened into a 2-D texture map, but the same flattening could also create an atlas by directly flattening the original mesh. Texture atlases do not require the continuity and smooth differentiability that good parameterization strive for.

Texture atlases have strived instead to minimize the distortion of its charts, and to minimize areas of discontinuity between chart images. Section 3.1 shows that distortion does not affect the quality of our method. Section 3.2 describes how discontinuities can cause seam artifacts, but we eliminate these artifacts later in Section 4.1. We instead offer two new measures of atlas quality: coverage (Sec. 3.3) and relative scale (Sec. 3.4), that are used to indicate the sampling fidelity offered by the atlas. Section 4 proposed new atlas techniques that perform well with respect to these two new measures.

3.1 Distortion

The *distortion* of a texture mapping is responsible for the deformation of a fixed image as it is mapped onto a surface.

[1] To keep these two textures straight, we will use $\mathbf{s} = (s,t,r)$ to indicate the *solid texture* coordinates and $\mathbf{u} = (u,v)$ to indicate the *texture map* coordinates. We will need to assign both kinds of coordinates to the vertices of a mesh.

[2] In topology, the atlas is used to define manifolds. In this context the atlas need not be one-to-one and the range of its charts may overlap.

Previous techniques for creating atlases have focused on reducing the distortion of the charts [43], either by projection [1], deformation energy minimization [20][21][22], or interactive placement [33][34].

Chart images are often complex polygons, and must then be packed (without further distortion) efficiently into the texture map to construct the atlas. Automatic packing methods for complex polygons are improving [25], but have not yet surpassed the abilities of human experts in this area.

Our use of a texture atlas for solid texturing is not directly affected by chart distortion. Solid texture coordinates are properly interpolated across the chart image in the texture map regardless of the difference in shape between the model-coordinate and the surface-texture-coordinate triangles. Chart distortion affects only the direction, or "grain" of the artifacts, but not their existence, as will be shown later in Figure 6.

3.2 Discontinuity

Texture atlases are discontinuous along the boundaries of their charts. Texture mapping can reveal these discontinuities as a rendering artifact known as a *seam*. Seams are pixels in the texture map along the edges of charts. They appear along the mesh edges as specks of the wrong color, either the texture map's background color or a color from a different part of the texture.

Previous techniques have reduced seams by maximizing the size and connectivity of the chart images in the texture atlas. For example, Maillot *et al.* [22] merged portions of the surface of similar curvature. These partitions improved the atlas continuity, resulting in fewer charts, though with complex boundaries. While this method reduced seams to the complex boundaries of fewer charts, it did not eliminate them.

Seams appear because the rasterization rules differ from texture magnification rules. The rules of polygon scan conversion are designed with the goal of plotting each pixel in a local polygonal mesh neighborhood only once[3]. The rules for texture magnification are designed to appropriately sample a texture when the sample location is not the center of a pixel, usually nearest neighbor or a higher order interpolation of the surrounding pixels.

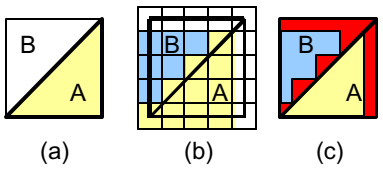


Figure 2. Seams occur due to differences between texture magnification (a) and rasterization (b), shown in red (c).

Figure 2a shows two triangles with integer coordinates in the texture map. Figure 2b shows these two triangles rasterized using the standard rules [7], with unrasterized white pixels in the background. In this figure, the integer pixel coordinates occur at the center of the grid cells. Hence the grid cell indicates the set of points whose nearest neighbor is the pixel located at the cell's center. Figure 2b illustrates that some points in both triangles A and B have background pixels as nearest neighbors, and some points in triangle B have pixels rasterized as triangle A because triangle A's pixels are their nearest neighbors. Figure 2c indicates these points in red.

Higher order texture magnification, such as bilinear or bicubic can reduce but not eliminate the effect of background pixels, and actually exaggerate the problem along the shared edge between triangles A and B. A common solution is to overscan the polygons in the texture map, but surrounding all three edges of each triangle with a one-pixel safety zone wastes valuable texture samples.

3.3 Coverage

The *coverage* C of an atlas measures how effectively the parameterization uses the available pixels in the texture map. The coverage ranges between zero and one and indicates the percentage of the texture map covered by the image of the mesh faces

$$C = \sum_{j=1}^{M} A(\mathbf{u}_{j1}, \mathbf{u}_{j2}, \mathbf{u}_{j3}) \qquad (1)$$

where $A()$ returns the area of a triangle. We assume the texture map is a unit square.

The coverage of atlases of packed complex polygons was quite low, covering less than half of the available texture samples in our tests. We also implemented a simple polygon packing method that used a single chart for each triangle. This triangle packing performed much better than the complex polygon packing, but still covered only 70% of the available texture samples. Since distortion does not affect the quality of our procedural solid texturing technique, the next section shows that the chart images of triangles can be distorted to cover most if not all of the available texture samples.

3.4 Relative Scale

Whereas the coverage measures how well the parameterization utilizes texture samples, the *relative scale* S indicates how evenly samples are distributed across the surface. We measure the relative scale as the RMS of the ratio of the square root of the areas before and after each chart of the atlas is applied

$$S^2 = \left(\sum_{j=1}^{M} A(\mathbf{x}_{j1}, \mathbf{x}_{j2}, \mathbf{x}_{j3}) \right) \frac{1}{M} \sum_{j=1}^{M} \frac{A(\mathbf{u}_{j1}, \mathbf{u}_{j2}, \mathbf{u}_{j3})}{A(\mathbf{x}_{j1}, \mathbf{x}_{j2}, \mathbf{x}_{j3})}. \qquad (2)$$

The additional summation factor computes the surface area of the object in model space, and normalizes the relative scale so it can be used as a measure to compare the quality of atlases across different models. A relative scale less than one indicates that the atlas is contracting a significant number of large triangles too severely, whereas a relative scale greater than one indicates that small triangles are taking up too large a portion of the texture map.

The relative scale of existing atlas techniques is typically less than one half. Inefficient packing yields low coverage, such that triangles must be scaled even smaller in order to make the complex chart images fit into available texture space.

4. Atlases for Solid Texturing

This section describes methods for constructing texture atlases specifically for procedural solid texturing that overcome sampling problems and seams.

[3] Missing pixels can result in holes or even cracks in the mesh, whereas plotting the same pixel twice (once for each of two different polygons) can cause pixel flashing as neighboring polygons battle for ownership of the pixel on their border.

4.1 Uniform Mesh Atlases

One way to take as many samples as possible is to maximize the coverage of texture map by the atlas. Since distortion does not affect the quality of the atlas for our application, we choose to deform the model triangles into a form that can be easily packed. The *uniform mesh atlas* arbitrarily maps all of the triangles into a single shape, an isosceles right triangle. These right triangles are packed into horizontal strips and stacked vertically in the texture map.

Figure 3 demonstrates the uniform mesh atlas. Continuity is ignored and the texture map can be thought of as a collection of rubber jigsaw puzzle pieces that must be stretched into an appropriate place on the model surface.

The length of each adjacent edge of the mesh triangles is given by

$$a = \frac{\lfloor \sqrt{M/2} \rfloor}{H} \qquad (3)$$

where H is the horizontal resolution of a square texture map. The floor ensures that we can plot a full row of triangle pairs. Note that a is not an integer, but non-integer edge lengths can create problems with seams.

Seam Elimination. Seams can be avoided by the careful rasterization of mesh triangles. Triangles A and B have been rasterized into the texture map as shown before. The triangles in Figure 4b are rasterized with half pixel offsets such that no background pixels will be accessed by the texture's magnification filter. Nonetheless, samples in triangle B near its hypotenuse will still return A's color. Overscanning the hypotenuse of triangle B and shifting triangle A right one pixel, as shown in Figure 4c, eliminates the seam artifact between A and B. This overscanning solution reduces the coverage slightly, but only costs one column of pixels for each triangle pair in a horizontal strip.

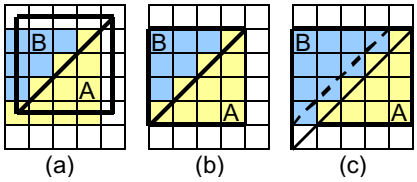


Figure 4. Standard rasterization rules disagree with texture magnification rules (a) and (b). Overscanned polygons are sampled correctly (c).

Since seams are eliminated, triangles can be placed in any order in the uniform mesh atlas. If the model contains triangle strips, then these strips can be inserted directly into the uniform mesh atlas without overscanning, as the edge they share has appropriate pixels on either side of it.

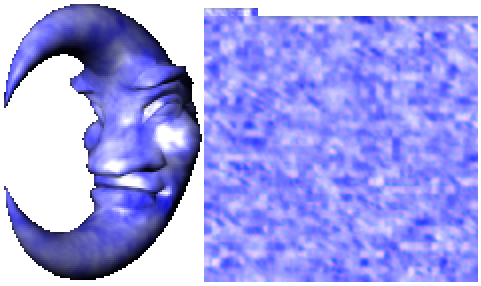


Figure 3. Uniform mesh atlas for a cloud textured moon.

4.2 Non-Uniform Mesh Atlases

While the uniform mesh atlas does a good job of using available texture samples, it distributes those samples unevenly. Object polygons both large and small get the same number of texture samples. The uniform mesh atlas biases the sampling of texture space in favor of areas with small triangles. While smaller polygons may appear in more interesting areas of the model, geometric detail might not correlate with texture detail.

Our goal is to not only use as many samples of the texture as possible, but to distribute those samples evenly across the model. The *non-uniform mesh atlas* attempts to more evenly distribute texture samples by varying the size of triangle chart images in the texture map.

Area-Weighted Mesh Atlas. An obvious criterion is that larger model triangles should receive more texture samples, and so their image under the atlas should be larger. We implement this *area-weighted* NUMA by first sorting the mesh triangles by non-increasing area. The mesh atlas is again constructed in horizontal strips, but the size of the triangles in the strip is weighted by the inverse of the relative scale of the triangles in the strip. This allows larger triangles to get more texture samples. Figure 5 demonstrates the area-weighted atlas on a rhino model.

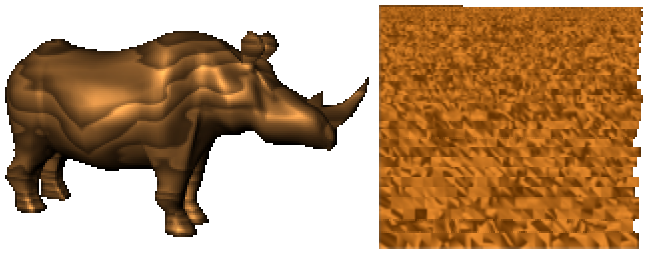


Figure 5: Rhino sculpted from wood and its area-weighted non-uniform mesh atlas.

Length-Weighted Mesh Atlas. Skinny triangles occupy smaller areas, but require extra sampling in their principal axis direction to avoid aliases. The *length-weighted* NUMA uses the triangle's longest edge to prioritize its space utilization in the texture map.

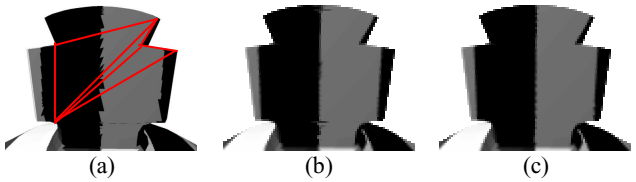


Figure 6. Effects of mesh atlas sample distribution techniques on a poorly tesselated object containing slivers: uniform (a), area weighted (b) and length weighted (c).

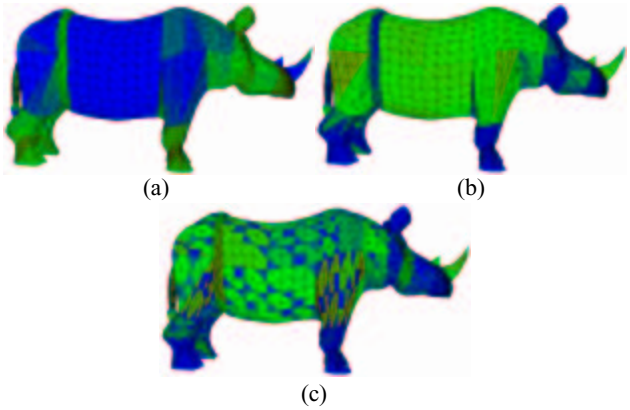
Figure 6 demonstrates the appearance of artifacts from the mesh atlases on the cross of a chess king piece. The procedural texture in this example is a simple striped pattern. Every triangle in the uniform mesh atlas (a) gets the same number of texture samples, regardless of size, resulting in the jagged sampling of the textured stripe on the left. The area-weighted NUMA reduces these aliasing artifacts, stealing extra samples from the rest of the model's smaller triangles. But the sliver polygon needs more samples than its area indicates, and the length-weighted NUMA gives the sliver triangles the same weight as their neighbors, reducing the aliasing completely, leaving only the artifacts of the nearest-neighbor texture magnification filter.

Comparison. We plotted the relative scale of each triangle in the meshed rhino model. The ideal relative scale is equal to the square root of the surface area, and is plotted in green. Since all of the uniform mesh atlas's chart image triangles are the same size, the plot of its relative scale simply indicates the size of the triangle in the model. Hence larger triangles are sample starved, but as Table 1 shows, a larger number of smaller triangles are receiving too many samples.

Mesh Atlas	Coverage	Relative Scale
Uniform	91%	1.75
Area-Weighted	93%	0.66
Length-Weighted	93%	0.86

Table 1. Measurement of mesh atlas performance on the rhino model.

The area-weighted mesh atlas does a much better job of distributing the samples, and nearly complements the sampling of the uniform mesh atlas. The area-weighted NUMA undersamples smaller triangles because they are assigned to the remaining scraps of the texture map, which also results in its relative scale of less than (but closer to) one.



(a) (b)

(c)

Figure 7. The rhino model color coded by the relative scale of each triangle under the uniform (a), area-weighted (b) and length-weighted (c) atlases. Green indicates optimal sampling, blue indicates too few samples, and red indicates too many.

Figure 7 illustrates the difference with this weighting, increasing the samples in the belt of skinny triangles around the rhino's waist, and the stretched triangles around its shoulder, by sacrificing some of the samples in the rest of the model. The length-weighting heuristic also improves the performance statistics, resulting in a relative scale much closer to the goal of one.

4.3 Multiresolution Mesh Atlases

Section 4.1 described how seam artifacts were removed by making rasterization agree with texture magnification. Texture minification also produces artifacts, aliasing when projected texture resolution exceeds screen resolution.

The MIP-map is a popular method for inhibiting texture minification aliases [46]. The MIP-map creates a multiresolution pyramid of textures, filtering the texture from full resolution in half-resolution steps down to a single pixel. Each pixel at level l of a MIP-map represents 4^l pixels of the full resolution texture map (at level 0).

Assume we have a uniform mesh atlas where the adjacent edge a of each of the triangles is a power of two. Then at levels up to $l_a = \lg a$, some pixels from both sides of a triangle pair will combine into a single pixel. This averaging is correct only if the triangle pair also shares an edge in the surface mesh.

At level $l_a + 1$, four neighboring triangle-pairs in the texture map will be averaged together. The uniform mesh atlas cannot be MIP-mapped at level l_a, + 1 or above as there is no spatial relationship between triangles in the atlas. We can however impose a spatial relationship on the uniform mesh atlas that permits MIP-mapping above level l_a.

At level l_a, triangle pairs are each represented by a single pixel. At level $l_a + 1$, the result of averaging neighboring triangles pairs is a single pixel. Hence, the mesh needs to have neighborhoods of triangle pairs grouped together, but the grouping need not be in any particular order.

We achieve this grouping by partitioning the surface mesh hierarchically into a balanced quadtree. Each level of the quadtree partitions the mesh into disjoint contiguous sections with (approximately) the same number of faces.

We implement our face partitioning using a multiconstraint-partitioning algorithm [18]. Such algorithms have found a wide variety of applications in computer graphics, e.g. [9][17][19].

The face hierarchy is constructed using the dual of the mesh. The partitioning algorithm uses edge collapses to repeatedly simplify this dual graph, yielding a hierarchy. The "balanced first choice" [18] heuristic is used to balance the hierarchy during simplification. We then optimize this graph from the top down, exchanging subtrees to minimize the edge length of the boundaries of the partitions. The result is demonstrated in Figure 8.

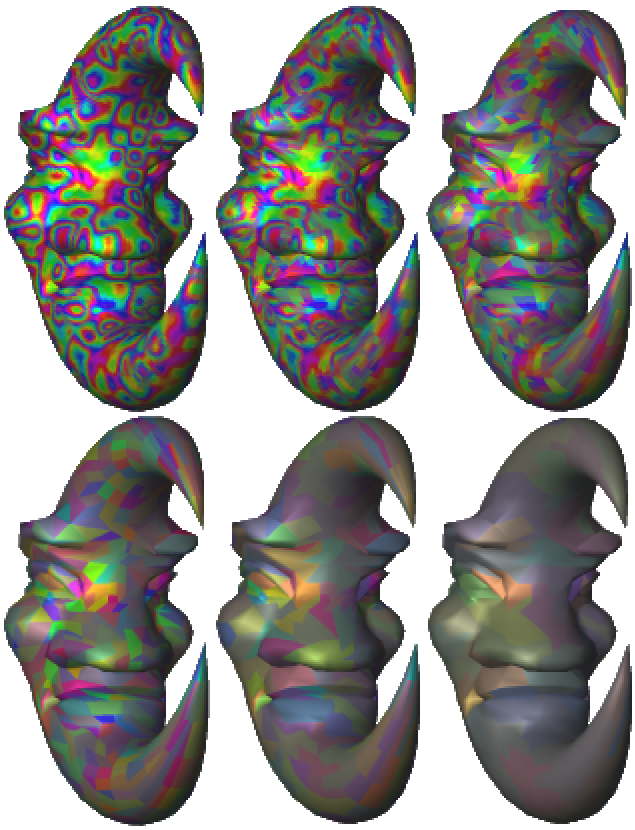


Figure 8. Levels of texture detail in the multiresolution uniform mesh atlas.

5. Procedural Texturing onto the Atlas

The solid texture coordinates resulting from the mesh atlases provides an efficient and direct method for applying procedural textures to an arbitrary object. We apply procedures directly to the texture map using the texture map containing solid texture coordinates interpolated across the polygon faces as input, replacing these coordinates with colors producing a texture map that when applied yields a procedural solid texturing of the object.

Procedural textures can be generated a number of ways. We explore two basic techniques. The first technique runs a procedure sequentially on the host. The second technique compiles the procedure into a multipass program executed in SIMD fashion by the graphics controller. We will focus on the Perlin noise function [37] as this single function is a widely used element of a large portion of procedural textures.

5.1 Host Rasterization

The texture atlas technique allows the procedural texture to be generated from the host. Host procedures provide the highest level of flexibility, allowing all of the benefits of a high-level language compiled into a broad instruction set.

Several fast host-processor methods exist for synthesizing procedural textures. Goehring *et al.* [10] implemented a smooth noise function in Intel MMX assembly language, evaluating the function on a sparse grid and using quadratic interpolation for the rest of the values. Kameya *et al.* [14] used streaming SIMD instructions that forward differenced a linearly interpolated noise function for fast rasterization of procedurally textured triangles.

One could use the graphics processor to rasterize the texture atlas, and then let the host processor replace the interpolated solid coordinates with procedural texture colors. The main drawback to this technique is the asymmetry of the graphics bus, which is designed for high speed transmission from the host to the graphics card. The channel from the graphics card to the host is very slow, taking nearly a second to perform an OpenGL ReadPixels command on an Intel PC AGP bus.

To overcome this bottleneck, our host-procedure implementation uses the host to rasterize the atlas directly into the texture map. Host rasterization provides full control over the rasterization rules and full precision for the interpolated texture coordinates. While the host processor is not nearly as fast as the graphics processor at rasterization, the generation and rendering of the atlas into texture memory is an interactive-time operation, whereas examination of the object is a real-time operation supported completely by the graphics card's texture mapping hardware. Its results are shown later in Table 3.

5.2 A Multipass Noise Algorithm

Following [15][23][35][41], we can harness the power of graphics accelerators to generate procedural textures directly on the graphics board.

The noise function could be implemented using a 3-D texture of random values with a linear magnification filter. A texture atlas of solid texture coordinates can be replaces with noise samples using the OpenGL pixel texture extension [31].

The vertex shader programming model found in Direct3D 8.0 [24] and the recent NVIDIA OpenGL vertex shader extension [31] can support procedural solid texturing. In fact a Perlin noise function has been implemented as a vertex program [29]. But a per-vertex procedural texture will produce vertex colors that are Gouraud interpolated across faces.

```
Input: solid_map with R,G,B containing s,t,r coordinates.
Initialize noise = black
solid_int = solid_map >> b_f
solid_intpp = solid_int + 1/(2^b-1)
weight = (solid_map – (solid_int << b_f)) << b_i
for (k = 0; k < 8; k++) {
  corner = solid_int
  corner = solid_intpp with glColorMask(k&1,k&2,k&4)
  randomize corner
  corner *= if (k&1) then R(weight) else 1 – R(weight)^4
  corner *= if (k&2) then G(weight) else 1 – G(weight)
  corner *= if (k&4) then B(weight) else 1 – B(weight)
  noise += corner
}
Output: solid noise texture map
```

Figure 9. Multipass noise algorithm.

We instead implemented a per-pixel noise function using multipass rendering onto the texture atlas. Assume the three channels (R,G,B) of our buffers have a depth of b bits[5]. We will assume a fixed-point representation with b_i integer bits and b_f fractional bits, $b = b_i + b_f$. The algorithm in Figure 9 computes a random value in [0,1] at the integer lattice points, and linearly interpolates these random values across the cells of the lattice.

SGI Implementation. We implemented the noise function in multipass OpenGL on imaging workstations using the glPixelTransfer and glPixelMap functions. The glPixelTransfer function performs a per-component scale and bias, whereas glPixelMap performs a per-component lookup. The results appear in Table 2.

NVidia Implementation. We also implemented a noise function for consumer-level accelerators using the NVidia chipset. Since the NVidia driver did not accelerate glPixelTransfer and glPixelMap, we used register combiners to shift, randomize and isolate/combine components.

Randomization on the NVidia controller was particularly difficult, as its driver did not accelerate logical operations like exclusive-or on the frame buffer. Instead, we used the register combiners to display one of two colors depending on an input color's high bit, then used the register combiners to shift the input color left one bit (without overflowing and causing a clamp to one). This ended up generating 375 passes (!). The source code for these operations can be found on the accompanying CD-ROM.

Implementation	Execution Time
SGI Solid Impact	1.3 Hz
SGI Octane	2.5 Hz
NVidia GeForce 256	0.9 Hz

Table 2. Execution times for the multipass noise algorithm.

Table 2 shows the NVidia implementation did not perform as well as the SGI implementation. Profiling the code revealed that the main bottleneck was the time it took to save the framebuffer in a texture, adding an average of 3 ms per pass for 354 of the passes. OpenGL currently does not support rendering directly to texture, and the register combiner did not directly support the blending of its output with the destination pixel currently in the frame buffer.

[4] The functions R(), G() and B() return a luminance image of the channel.

[5] Framebuffers currently hold only 8 or 12 bits per channel though there is an extension that supports 32-bit floating point, and indications that floating point buffers may soon be supported by a larger variety of graphics hardware and drivers.

The randomization step in the SGI implementation produced white noise using a glPixelMap lookup table of random values, whereas the NVidia implementation blended random colors, yielding Gaussian noise. If desired, one could redistribute the Gaussian noise into white noise with a fixed histogram equalization step.

6. Conclusion

We have shown how the texture atlas can facilitate the real-time application of solid procedural texturing. We showed that for this application, the texture atlas need not be concerned with distortion nor discontinuity, but should instead focus on sampling fidelity. We introduced new mesh-based atlas generation schemes that more efficiently used available texture samples, and non-uniform variations of these meshes distributed these samples more evenly across the object. We also used a mesh partitioning method to construct a MIP-mappable atlas.

The texture atlas allows solid texturing procedures to be applied to the texture map, allowing efficient multipass programming using the accelerated operations available on the graphics controller as they become feasible.

The system makes effective use of preprocessing. The procedural texture needs to be resynthesized only when its parameters change, and the texture atlas needs to be reconstructed only when the object changes shape. Specifically, if the position of the object's vertices move, but the topology of the mesh remains invariant, then the procedural solid texturing generated by this method will adhere to the surface [1]. This is a useful property that prevents texture "swimming," such that for example the grain of a warped wood plank follows the warp of the plank.

6.1 Interactive Procedural Solid Texture Design

We used the methods described in this paper to create a procedural solid texture design system that would allow the user to load an object and apply a procedural solid texture. This system can be found on the accompanying CD-ROM. Since the procedural solid texturing is applied as a standard 2-D surface texture mapping, the design system supported full real-time observation of a procedurally solid textured object. Using the techniques of Section 4, the object did not suffer from any seam artifacts, and aliasing was reduced by making good use of the available texture samples.

We also allowed the user to interactively change the procedural solid texturing parameters. Using the techniques described in Section 5.1, we were able to support interactive-rate feedback to the user, such that the user could observe the result of a parameter on the procedural solid texture while dragging a slider.

The software procedural texture renderer simultaneously rasterized the texture atlas into texture memory and applied the texturing procedure to the texture atlas. We increased the responsiveness of our system by having this renderer render a lower resolution interpolated version of the atlas during manipulation, and replace it with a higher resolution version at rest. The rendering speed of this system is shown in Table 3.

Noise Octaves	Atlas Res.	Procedural Synthesis Speed
1	256^2	9.09 Hz (18 Hz)
1	512^2	2.56 Hz (4.55 Hz)
1	1024^2	0.72 Hz (1.30 Hz)
4	256^2	6.25 Hz (10 Hz)
4	512^2	1.82 Hz (3.03 Hz)
4	1024^2	0.40 Hz (0.76 Hz)

Table 3. Execution times for procedural texture synthesis into the texture atlas. Parenthetic times measure lower resolution synthesis during interaction.

6.2 Applications

We have focused this paper on the application of real-time procedural solid texturing, though the techniques described appear to impact other areas as well.

Solid Texture Encapsulation. Unlike surface texture coordinates, solid texture coordinates are not uniformly implemented by graphics file formats. Using surface texture of a solid texture allows the texture coordinates to be more robustly specified in object files and also allows the solid texture to be included as a more compact texture map image instead of a wasteful 3-D solid texture array.

3-D Painting. The meshed atlas techniques can also be used to support 3-D painting onto surfaces [13]. The atlas provides an automatic parameterization. The discontinuities of the parameterization do not impact painting as the texture atlas maintains a per face correspondence between the surface and the texture map. The meshed atlas techniques presented in Section 4 also improve surface painting by using as many texture samples as possible distributed evenly across the surface.

Normal Maps. The normal map [3][8] is a texture map whose pixels hold a surface normal instead of a color. Normal maps are used for real-time per-pixel bump mapping using dot-product texture combiners found in Direct3D and extensions of OpenGL. The meshed atlas generation techniques can be used to create well-sampled normal maps since normal maps do not require continuity between faces.

Real-Time Shading Languages. Recent real time shading languages [35][41] have been developed to support procedural shaders, including texturing and lighting, by converting shader descriptions into multipass graphics library routines. In particular, Proudfoot *et al.* [41] focuses on the difference between per object, per vertex and per fragment processes in real-time shaders. The texture atlas supports additional categories of view-dependent and view-independent processes. View dependent processes utilize multipass operations to the framebuffer, whereas view independent processes utilize multipass operations to the texture map, ala Section 5.2. The results of view independent processes can be stored and accessed directly from the texture map, accelerating the rendering of real time shading language shaders.

6.3 Future Work

While this work achieved our goal of real-time procedural solid texturing, it has also inspired several directions for further improvement.

Direct Manipulation of Procedural Textures. The interactive procedural solid texture design system is a first step. Another step would be to allow the sliders to be bypassed, supporting direct manipulation of procedural textures. The user could drag a texture feature to a desired location and have the software automatically reconfigure the parameters appropriately.

Preservation of Mesh Structure. The mesh atlases do not preserve the object's original mesh structure, and our mesh atlas processing program outputs multiple copies of shared mesh vertices with different surface texture coordinates. This increases the size of the model description files, and may cause the resulting models to render more slowly. Preservation of mesh structure, or at least triangle strips, would be a useful addition to this stage of the process.

Higher-Order Texture Magnification. Section 4.1 described the special overscanning measures taken during rasterization of the texture atlas to eliminate seam artifacts. This overscanning works when a nearest neighbor texture magnification filter is used. A

linear texture magnification filter would make the textures appear less blocky, but will require overscanning by one pixel along all edges reduces the number of available samples on polygon faces creating additional seldom used samples on polygon edges.

Atlas Compression. The texture atlas resembles the codebook used in vector quantization. The number of faces in the atlas could be reduced by allowing the atlas to no longer be one-to-one, and to let triangles with similar procedural texture features to map to the same location in the texture atlas. This kind of atlas compression would increase the number of available texture samples with larger chart images in the texture atlas.

6.4 Acknowledgments

This research was funded in part by the Evans & Sutherland Computer Corp. overseen by Peter K. Doenges. The research was performed using facilities at both Washington State University and the University of Illinois. Jerome Maillot was instrumental in showing us the state of the art in this area, including Alias|Wavefront's work. Pat Hanrahan observed that the UMA biases the MIP map in favor of smaller triangles.

References

[1] Apodaca, A.A. Advanced Renderman: Creating CGI for Motion Pictures. Morgan Jaufmannm 1999. See also: Renderman Ticks Everyone Should Know, in SIGGRAPH 98 or SIGGRAPH 99 Advanced Renderman Course Notes.

[1] Bennis, C. J Vezien, and G. Iglesias. Piecewise surface flattening for non-distorted texture mapping. *Proc. SIGGRAPH 91*, July 1991, pp. 237-246.

[2] Brinsmead, D. Convert solid texture. Software component of *Alias|Wavefront Power Animator 5*, 1993.

[3] Cohen, J., M. Olano and D. Manocha. Appearance-Preserving Simplification. *Proc. SIGGRAPH 98*, July 1998, pp. 115-122.

[4] Crow, F.C. Summed area tables for texture mapping. Computer Graphics 18(3), (*Proc. SIGGRAPH 84*), July 1984, pp. 137-145.

[5] DoCarmo, M. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, 1976.

[6] Ebert, D., F.K. Musgrave, D. Peachey, K. Perlin and S. Worley. Texturing and Modeling: A Procedural Approach, Academic Press.1994.

[7] Foley, J.D., A. van Dam, S.K. Feiner and J.F. Hughes. Computer Graphics, Principles and Practice, Second Edition, Addison-Wesley, 1990.

[8] Fournier, A. Normal distribution functions and multiple surfaces. Graphics Interface '92 Workshop on Local Illumination, May 1992, pp. 45-52.

[9] Garland, M., A. Willmott and P.S. Heckbert. Hierarchical face clustering on polygonal surfaces. Proc. Interactive 3D Graphics, March 2001, To appear.

[10] Goehring, D. and O. Gerlitz. Advanced procedural texturing using MMX technology. Intel MMX Technology Application Note, Oct. 1997. http://developer.intel.com/software/idap/ resources/technical_collateral/mmx/proctex2.htm

[11] Hanrahan, P. and J. Lawson. A language for shading and lighting calculations. *Computer Graphics* 24(4), (*Proc. SIGGRAPH 90*), Aug. 1990, pp. 289-298.

[12] Hanrahan, P. Procedural shading (keynote). Eurographics / SIGGRAPH Workshop on Graphics Hardware, Aug. 1999. http://graphics.standford.edu/hanrahan/talks/rts1/slides.

[13] Hanrahan, P. and P.E. Haeberli. Direct WYSIWYG Painting and Texturing on 3D Shapes, *Computer Graphics 24* (4), (*Proc. SIGGRAPH 90*), Aug. 1990, pp. 215-223.

[14] Hart, J. C., N. Carr, M. Kameya, S. A. Tibbits, and T.J Colemen. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Aug. 1999, pp. 45-53.

[15] Heidrich, W. and H.-P. Seidel. Realistic hardware-accelerated shading and lighting. *Proc. SIGGRAPH 99*, Aug. 1999, pp. 171-178.

[16] Kameya, M. and J.C. Hart. Bresenham noise. *SIGGRAPH 2000 Conference Abstracts and Applications*, July 2000.

[17] Karni, Z. and C. Gotsman. Spectral compression of mesh geometry. *Proc. SIGGRAPH 2000*, July 2000, pp. 279-286.

[18] Karypis, G. and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. *Proc. Supercomputing 98*, Nov. 1998.

[19] Lee, A.W.F., W. Sweldens, P. Schröder, L. Cowsar, D. Dobkin. MAPS: Multiresolution Adaptive Parameterization of Surfaces. *Proc. SIGGRAPH 98*, July 1998, pp. 95-104.

[20] Levy, B. and J.L. Mallet. Non-distorted texture mapping for sheared triangulated meshes. *Proc. SIGGRAPH 98*, July 1998, pp. 343-352.

[21] Ma, S. and H. Lin. Optimal texture mapping. *Proc. Eurographics '88*, Sept. 1988, pp. 421-428.

[22] Maillot, J., H. Yahia and A. Verroust. Interactive texture mapping. *Proc. SIGGRAPH 93*, Aug. 1993, pp. 27-34.

[23] McCool, M.C. and W. Heidrich. Texture Shaders. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Aug. 1999, pp. 117-126.

[24] Microsoft Corp. Direct3D 8.0 specification. Available at: http://www.msdn.microsoft.com/directx.

[25] Milenkovic, V.J. Rotational polygon overlap minimization and compaction. *Computational Geometry: Theory and Applications* 10, 1998, pp. 305-318.

[26] Molnar, S., J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. *Computer Graphics 26*(2), (*Proc. SIGGRAPH 92*), July 1992, pp. 231-240.

[27] Munkres, J.R. Topology; A First Course. Prentice Hall, 1974.

[28] Norton, A., A.P. Rockwood, and P.T. Skolmoski. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. *Computer Graphics* 16(3), (*Proc. SIGGRAPH 82*), July 1982, pp. 1-8.

[29] NVidia Corp. Noise, component of the NVEffectsBrowser. Available at: http://www.nvidia.com/developer.

[30] Olano, M. and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. Proc. SIGGRAPH 98, July 1998, pp. 159-168.

[31] OpenGL Architecture Review Board. OpenGL Extension Registry. Available at: http://oss.sgi.com/projects/ogl-sample/registry/

[32] Peachey, D.R. Solid texturing of complex surfaces. *Computer Graphics* 19(3), July 1985, pp. 279-286.

[33] Pedersen, H.K. Decorating implicit surfaces. *Proc. SIGGRAPH 95*, Aug. 1995, pp. 291-300.

[34] Pedersen, H.K. A framework for interactive texturing operations on curved surfaces. *Proc. SIGGRAPH 96*, Aug. 1996, pp. 295-302.

[35] Peercy, M.S., M. Olano, J. Airey and P.J. Ungar. Interactive multi-pass programmable shading, Proc. SIGGRAPH 2000, July 2000, pp. 425-432.

[36] Perlin, K and E.M. Hoffert. Hypertexture. *Computer Graphics* 23(3), July 1989, pp. 253-262.

[37] Perlin, K. An image synthesizer. *Computer Graphics* 19(3). July 1985, pp. 287-296.

[38] Pixar Animation Studios. Future requirements for graphics hardware. Memo, 12 April 1999.

[39] Potmesil, M., and E.M. Hoffert. The Pixel Machine: A parallel image computer. *Computer Graphics 23*(3), (Proceedings of SIGGRAPH 89), July 1989, pp. 69-78.

[40] Praun, E., A. Finkelstein and H. Hoppe. Lapped Textures, *Proc. SIGGRAPH 2000*, July 2000, pp. 465-470.

[41] Proudfoot, K., W.R. Mark and Pat Hanrahan. A framework for real-time programmable shading with flexible vertex and fragment processing. Manuscript, Jan. 2000. See also: http://graphics.stanford.edu/projects/shading.

[42] Rhoades, J., G. Turk, A. Bell, U. Neumann, and A. Varshney. Real-time procedural textures. 1992 *Symposium on Interactive 3D Graphics* 25(2), March 1992, pp 95-100.

[43] Samek, M. Texture mapping and distortion in digital graphics. *The Visual Computer* 2(5), 1986, pp. 313-320.

[44] Segal, M. and K. Akeley. The OpenGL Graphics System: A Specification, Version 1.2.1. Available at: http://www.opengl.org/.

[45] Thorne, C. Convert solid texture. Software component of *Alias|Wavefront Maya 1*, 1997.

[46] Williams, L. Pyramidal parametrics. *Computer Graphics* 17(3), July 1983, pp. 1-11, Proc. SIGGRAPH 83.

[47] Wyvill G., B. Wyvill, and C. McPheeters. Solid texturing of soft objects. IEEE Computer Graphics and Applications 7(4), Dec. 1987, pp. 20-26.

Perlin Noise Pixel Shaders

John C. Hart
University of Illinois, Urbana-Champaign

Abstract

While working on a method for supporting real-time procedural solid texturing, we developed a general purpose multipass pixel shader to generate the Perlin noise function. We implemented this algorithm on SGI workstations using accelerated OpenGL PixelMap and PixelTransfer operations, achieving a rate of 2.5 Hz for a 256x256 image. We also implemented the noise algorithm on the NVidia GeForce2 using register combiners. Our register combiner implementation required 375 passes, but ran at 1.3 Hz. This exercise illustrated a variety of abilities and shortcomings of current graphics hardware. The paper concludes with an exploration of directions for expanding pixel shading hardware to further support iterative multipass pixel-shader applications.

Keywords: Pixel shaders, Perlin noise function, hardware shading, register combiners.

1. Introduction

The concept of procedural shading is well known [17][19], and has found widespread use in graphics [3]. Procedural shading computes arbitrary lighting and texture models on demand. Procedural textures efficiently support high resolution, non-repeating features indexed by three-dimensional solid texture coordinates. These features were quickly adopted for production-quality rendering by the entertainment industry, and became a core component of the Renderman Shading Language [5].

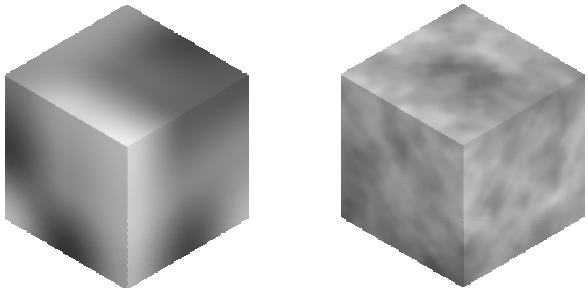
With the acceleration of graphics processors outpacing the exponential growth of general processors, there have been several recent calls for real-time implementations of procedural shaders, e.g. [6][20]. Real-time procedural shading makes videogames richer, virtual environments more realistic and modeling software more faithful to its final result. Real-time procedural texturing, in particular, allows modelers to use solid textures to seamlessly simulate sculptures of wood and stone. It yields complex animated environments with billowing clouds and flickering fires. Designers and users can interactively synthesize and investigate new procedural worlds that seem vaguely familiar to our own but with features unique to themselves.

Contact info: Dept. of Computer Science, 1304 W. Springfield Ave., Urbana, IL 61801, (217) 333-8740, jch@cs.uiuc.edu.

Several have researched techniques for supporting procedural shading with real-time graphics hardware [15][18][21][22]. These shading methods reorganize the architecture of the graphics API to suit the needs of procedural shading, applying API components to tasks for which they were not originally designed [8][11].

One such technique supports real-time procedural solid texturing [2] by using the texture map to store the shading of an object [1]. The technique maintains a texture atlas that maps triangles from a surface mesh into a non-overlapping array in texture memory. The triangles are plotted in texture memory using their solid texture coordinates as vertex colors. Rasterization then interpolates solid texture coordinates across their faces in the texture map. A procedural texturing pass replaces the solid texture coordinates in the texture map with the procedural texture color. Finally, this color is reapplied to the object surface via standard texture mapping. The result is a view-independent procedural solid texturing of the object.

One of the most common components of a procedural shading system is the Perlin noise function [19], a correlated three-dimensional field of uniform random values. This versatile function provides a deterministic random function whose bandwidth can be controlled to inhibit aliasing. Moreover, $1/f^{\beta}$ sums of noise functions can be used to form turbulence and other fractal structures whose statistics can be set to match those of various kinds of natural phenomena.



(a) (b)

Figure 1. Perlin noise function (a) and a $1/f$ sum (b).

We integrated the Perlin noise function into our real-time procedural solid texturing system in a variety of different ways, both as a CPU process and as a GPU process. This paper describes an algorithm for implementing the Perlin noise function as a multipass pixel shader. It also analyzes this noise implementation on a variety of systems. We used the available accelerated implementations of the OpenGL API and its device-dependent extensions on two SGI systems and an NVidia GeForce2. The paper concludes with suggestions for further

hardware accelerator development that would facilitate faster implementations of the Perlin noise function as well as a broader variety of texturing procedures.

2. Previous work

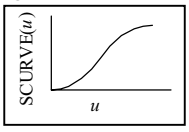
Because the Perlin noise function has become a ubiquitous but expensive tool in texture synthesis, it has been implemented in highly optimized forms on a variety of general and special purpose platforms.

Several fast host-processor methods exist for synthesizing Perlin noise. Goehring *et al.* [4] implemented a smooth noise function in Intel MMX assembly language, evaluating the function on a sparse grid and using quadratic interpolation for the rest of the values. Kameya *et al.* [10] used streaming SIMD instructions that forward differenced a linearly interpolated noise function for fast rasterization of procedurally textured triangles.

One can also generate solid noise with a 3-D texture array of random values [13], using hardware trilinear interpolation to correlate the random lattice values stored in the volumetric texture. Fractal turbulence functions can be created using multitexture/multipass modulate and sum operations. A texture atlas of solid texture coordinates would then be replaced with noise samples using the OpenGL pixel texture extension, ala [9].

The vertex-shader programming model found in Direct3D 8.0 [12] and the recent NVIDIA OpenGL vertex shader extension [16] can support procedural solid texturing. A Perlin noise function has been implemented as a vertex program [14]. But a per-vertex procedural texture produces vertex colors that are Gouraud interpolated across faces, such that the frequency of the noise function must be at, or less than half, the frequency of the mesh vertices. This would severely restrict the use of turbulence resulting from $1/f$ sums of noise. Hence the Perlin noise vertex shader is limited to low-frequency displacement mapping or other noise effects that can be mesh frequency bound.

Our favorite implementation of the Perlin noise function is from the Rayshade ray tracer [24]. This implementation created its own pseudorandom numbers by hashing integer solid texture coordinates with a scalar function Hash3d(i,j,k), then interpolated these random values with a simple smooth cubic interpolant SCURVE(u) = $3u^2 - 2u^3$ to yield the final result.



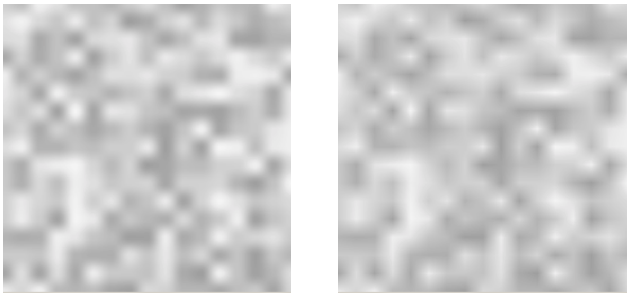
Given solid texture coordinates s,t,r, the Rayshade noise function effectively returned noise as the value

$$\sum_{k=0}^{1}\sum_{j=0}^{1}\sum_{i=0}^{1} \text{Hash3d}(\lfloor s \rfloor + i, \lfloor t \rfloor + j, \lfloor r \rfloor + k) w(s,i) w(t,j) w(r,k)$$

where

$$w(s,i) = \text{SCURVE}(s - \lfloor s \rfloor)^i \, (1 - \text{SCURVE}(s - \lfloor s \rfloor))^{1-i}$$

is a weighting function. Hence, the noise function returns a weighted sum of the random values at the eight corners of the integer lattice cube containing s,t,r.



(a) (b)

Figure 2. Result of the Rayshade implementation of the Perlin noise function, using cubic interpolation (a) and linear interpolation (b) of corner lattice random values.

Figure 2 demonstrates the result of the Rayshade implementation of the Perlin noise function. The random values result from the drand48() function of the standard C math library. Noise is defined on an integer coordinate lattice, which results in the strong horizontal and vertical correlation.

We will use this sample as a reference to compare our pixel-shader implementations of the Perlin noise function. The average brightness of the (s,t) slice of the noise is due to the fixed r coordinate. This average intensity will differ from across implementations, resulting in variations in brightness for a given (s,t) slice of the three-dimensional noise field.

3. A Multipass Noise Algorithm

We based our real-time implementation of the Perlin noise function on the concise Rayshade implementation. We implemented a per-pixel noise function using multipass rendering onto a texture atlas initialized with solid texture coordinates stored as pixel colors.

The Perlin noise function is defined on a field of real values, where the integer subset of its domain defines the base frequency of the noise. Implementation of the noise function requires coordinates s,t,r to range over multiple integers, though color components only range over [0,1]. Hence, given three channels (R,G,B) each with a depth of b bits[1], we use a fixed-point representation with b_i integer bits and b_f fractional bits, $b = b_i + b_f$.

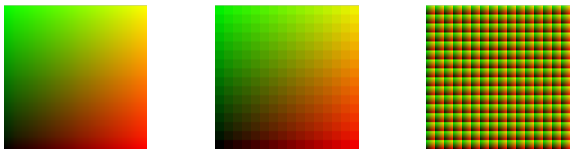
Following the form of the Rayshade noise implementation, the algorithm in Figure 3 computes a random value in [0,1] at the integer lattice points, and linearly interpolates these random values across the cells of the lattice.

[1] Framebuffers currently hold only 8 or 12 bits per channel though there is an extension that supports 32-bit floating point, and indications that floating point buffers may soon be supported by a larger variety of graphics hardware and drivers.

```
Input: 2-D texture solid_map with R,G,B containing s,t,r
coordinates.
Initialize texture noise = black
texture solid_int = solid_map >> b_f
texture solid_intpp = solid_int + 1/(2^b-1)
texture weight = (solid_map – (solid_int << b_f)) << b_i
for (k = 0; k < 8; k++) {
  texture corner = solid_int
  overwrite corner = solid_intpp with glColorMask(k&1,k&2,k&4)
  randomize corner
  corner *= if (k&1) then R(weight) else 1 – R(weight)^2
  corner *= if (k&2) then G(weight) else 1 – G(weight)
  corner *= if (k&4) then B(weight) else 1 – B(weight)
  noise += corner
}
Output: solid noise texture map
```

Figure 3. Multipass noise algorithm.

The input to the algorithm is an image solid_map whose R,G,B colors consist of solid texture coordinates. The first half of the algorithm decomposes solid_map into its integer part solid_int shifted right b_f times and a fractional part weight shifted left b_i times.



(a) (b) (c)

Figure 4. Solid texture coordinates solid_map (a), tex_int shifted left by b_f (b) and weight (fractional part shifted left by b_f) (c).

Figure 4 shows a sample texture map as a plane of two-dimensional solid texture coordinates spanned by s and t. We set b_f = 4 bits. The solid texture coordinates s,t,r range from (0.0,0.0,0.0) to (15.9375,15.9375,0.0) and are represented in the solid texture coordinate texture map Figure 4(a) with RGB colors from (0,0,0) to (1,1,0). Internally in the 24bpp framebuffer, these RGB colors range from (0,0,0) to (255,255,0). These coordinates are shifted right by b_f to form tex_int, which is shown Figure 4(b) shifted left by b_f to increase contrast and brightness. Subtracting (b) from (a) leaves tex_frac, which is shifted left by b_f to create a normalized weight function Figure 4(c).

The color (R,G,B) of each pixel (x,y) in solid_map corresponds to a solid texture point $(s=R,t=G,r=B)$ that falls within some lattice cell. The corner of this cell is given by the coordinates in the corresponding pixel (x,y) stored in solid_int. The opposite corner of this cell is found in the corresponding pixel in solid_intpp (whose colors are increments of those in solid_int).

Each of the eight corners of the cell can be found by combinations of the coordinates in solid_int and solid_intpp. The second half of the algorithm iterates over all eight corners, creating a random value indexed by the integer value at that corner. These random values are weighted by the fractional portion of the solid texture coordinates found in weight or its additive inverse. Summing the products of these weights for each of the eight corners performs a trilinear interpolation of the random values at the corners, resulting in result of the noise function.

We will spend the next two sections implementing this algorithm using the available accelerated features of two different graphics architectures. These implementations are each divided into two sections, on implementing the logical shift operations needed for the first half of the algorithm, and the random value synthesis needed for the second half.

4. SGI Implementation

The SGI graphics accelerators have focused on high-end real-time rendering for the scientific visualization and entertainment production communities. Hence accelerated features have included scientific imaging functions that support algebraic and lookup-table operations on pixels.

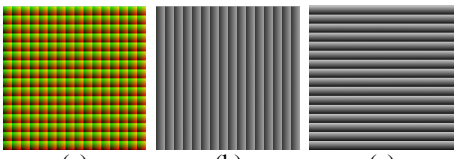
We focused our implementation on low end and midline SGI workstations, which are commonly deployed for digital content creation and design in both the videogame and animation communities.

4.1 PixelTransfer and PixelMap

We implemented the noise function in multipass OpenGL on SGI workstations using accelerated PixelTransfer[3] and PixelMap functions. The PixelTransfer function performs a per-component scale and bias, whereas PixelMap performs a per-component lookup into a predefined table of values.

We defined an assembly language of useful PixelTransfer functions. Specifically, the function setPixelTransfer(a,b) sets OpenGL to perform an $ax + b$ operation during the next image transfer operation, where x represents each component of the RGBA color. The function setPixelMap(table) uses PixelMap to replace colors channels with their corresponding entries in a lookup table. We also defined a blendtex(i) operation that draws the texture image corresponding to texture index i. The instruction savetex(i) saves the current framebuffer as texture image i.

Unlike the previous section, the SGI implementation begins with three luminance images tex_s, tex_t and tex_r instead of a single RGB image solid_map. We could perform all of the decompositions on a single texture, but we would later need to break its red, green and blue channels into individual luminance textures, and we found it impossible to perform this efficiently with the OpenGL extension set available to low-end and midline SGI workstations that lacked the color_matrix extension.



(a) (b) (c)

Figure 5. RGB image weight (a) is equal to (1,0,0) * luminance image tex_s (b) + (0,1,0) * luminance image tex_t (c) + (0,0,1) * luminance image tex_r (not shown).

[2] The functions R(), G() and B() return a luminance image of the corresponding channel.

[3] Following the convention of the OpenGL ARB, we avoid the use of the "gl" prefix for functions and the "GL_" prefix for tokens when describing elements of the OpenGL API.

4.2 Logical Shift Operations

The task of decomposing a texture map of fixed point solid texture coordinates into integer and fractional textures used PixelTransfer multiplication to achieve shifting operations. We defined an integer shift = $1 << b_f$. We modulated the texture by shift to perform a logical shift left by b_f, and by 1/shift to perform a logical shift right. (Some hardware required us to round instead of truncate, which was performed by a PixelTransfer bias of -0.5/255.0.) We also defined fracshift as $255.0/((1 << b_f) - 1)$. This allowed us to scale our fractional portions into normalized weights.

The following code fragment demonstrates the decomposition of the *s* coordinate. Similar decompositions need to be performed on tex_t and tex_r as well.

```
// shift s right to remove fractional part, save as si
blendtex(tex_s);
setPixelTransfer(1.0/shift, 0.0 /* or –0.5/255.0 */);
savetex(tex_si);
resetPixelTransfer();

// shift si back left
blendtex(tex_si);
setPixelTransfer(shift, 0.0);
CopyPixels(0,0,HRES,VRES,COLOR);
resetPixelTransfer();

// subtract si (floor of s) from s to get fractional part of s
Enable(BLEND);
BlendEquation(SUBTRACT);
BlendFunc(1, 1);
blendtex(tex_s);
Disable(BLEND);

// scale fractional part into normalized weight in [0,1]
setPixelTransfer(fracshift, 0.0);
savetex(tex_sf);
resetPixelTransfer();
```

4.3 Random Value Synthesis

We implemented randomization using a lookup table. This lookup table was accessed using the accelerated PixelMap OpenGL function. Recall the value k ranges from 0 to 7 denoting the current corner. The following code fragment synthesizes a random field based on the *s* coordinate.

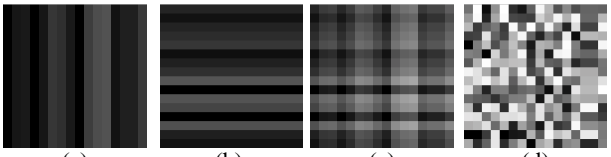
```
// tex_sin = random(si) or random(si++)
blendtex(tex_si);
setPixelTransferf(1.0, (k&1) ? 1.0/255.0 : 0.0);
setPixelMap(sran);
savetex(tex_sin);
```

Similar code fragments apply to the *t* and *r* coordinates, using (k&2) and (k&4) in the PixelTransfer, respectively. At this point tex_sin, tex_tin and tex_rin contain random values indexed by the *s*,*t*,*r* values at the *k*th corner of the cell. The following code fragment combines these three random values into a single random value.

```
// now tex_sin, tex_tin and tex_rin are random
// add them up into a single random number[4]
blendtex(tex_sin);
Enable(BLEND); BlendFunc(ONE,ONE);
blendtex(tex_tin);
blendtex(tex_rin);
Disable(BLEND);
```

This combination of random values is highly correlated due to the componentwise combination of random values. We reduce this correlation with an additional randomization pass.

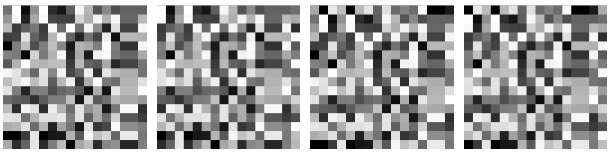
```
// one more randomization (in place)
setPixelMap(nran);
CopyPixels(0,0,HRES,VRES,COLOR);
resetPixelTransfer();
```



(a) (b) (c) (d)

Figure 6. The sum of random numbers indexed by *s* (a). and *t* (b) is highly correlated (c). This correlation is reduced by indexing into a final randomization (d).

The random number tables sran, tran and rran are uniform random number distributions over the range [0,1/3]. These three random values are added to form the final distribution, which is slightly non-uniform and heavily coordinate correlated, as shown in Figure 6(c). An additional randomization reduces this correlation as shown in Figure 6(d).



(a) (b) (c) (d)

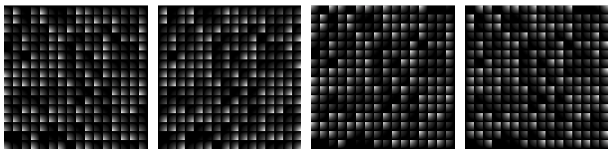
Figure 7. The random values at integer lattice locations for corners ($\lfloor s \rfloor,\lfloor t \rfloor$) (a), ($\lfloor s \rfloor+1,\lfloor t \rfloor$) (b), ($\lfloor s \rfloor,\lfloor t \rfloor+1$) (c) and ($\lfloor s \rfloor+1,\lfloor t \rfloor+1$) (d).

Figure 7 shows the random values generated at the four corners of the lattice. Note that in this example these are all translates of each other.

The random value is then weighted by the fractional part of the original texture coordinates s,t,r. Note that we have broken out the original RGB image weight from the previous section into three luminance images tex_sf, tex_tf and tex_rf. We also use the built-in additive complement blending operation to invert the weight appropriately depending on the cell corner.

```
// displayed texture now random value at corner k
// weight this contribution by fractional parts of s,t,r
Enable(BLEND);
BlendFunc(0, (k&1) ? SRC : 1 - SRC);
blendtex(tex_sf);
blendFunc(0, (k&2) ? SRC : 1 - SRC);
blendtex(tex_tf);
BlendFunc(0, (k&4) ? SRC : 1 - SRC);
blendtex(tex_rf);
```

[4] Note the addition of the component random values introduces a slight Gaussian bias to the resulting noise. This could be eliminated if an accelerated exclusive-or blending mode was available.



(a) (b) (c) (d)

Figure 8. Random values scaled by the weight functions (1 - tex_sf)(1-tex_tf) (a), tex_sf(1-tex_tf) (b), (1-tex_sf)tex_tf (c) and tex_sf tex_tf (d).

Figure 8 shows the random values at the corners (Figure 7) scaled by the product of weighting functions tex_sf and tex_tf. These weighting functions are luminance textures corresponding to the individual channels of Figure 4(c), such that weight = (tex_sf, tex_tf, tex_rf).

The resulting weighted random value corresponding to the current corner is then added into a running total, as show in the following fragment.

```
// add noise component into noise sum
BlendFunc(1,1);
blendtex(tex_noise);
Disable(BLEND);

// keep track of sum
savetex(tex_noise);
```

The texture tex_noise is initialized to black. After all eight corners have been visited, tex_noise contains the final noise values corresponding to the solid texture coordinates in the input luminance images tex_s, tex_t and tex_r.

Figure 9. Noise function resulting from the sum of Figure 8 (a-d).

4.4 Results

Figure 9 shows the final noise function resulting from summing the images in Figure 8. The correlation from Figure 6(c) was reduced by the randomization in Figure 6(d) but is still evident, particularly in the final interpolated version, as strong horizontal and vertical tendencies in the noise. However, this correlation is also found in the reference noise implementation in Figure 2, and is primarily due to the integer lattice of noise values.

We implemented this algorithm at a resolution of 256^2 on a SGI Solid Impact, a SGI Octane, and an NVidia GeForce2. The SGI workstations are designed for advanced imaging applications and have hardware accelerated PixelTransfer and PixelMap operations whereas the NVidia card designed for mainstream consumer applications does not. The execution times are given in Table 1.

Implementation	Execution Time (Rate)
SGI Octane	0.4 sec. (2.5 Hz)
SGI Solid Impact	0.75 sec. (1.3 Hz)
NVidia GeForce 256	5 sec. (0.2 Hz)

Table 1. Execution results for the multipass noise algorithm.

5. NVidia Implementation

We also implemented a noise function for consumer-level accelerators using the NVidia chipset. The NVidia products have been designed to accelerate commodity personal computer graphics, especially videogames. Hence the drivers did not accelerate PixelTransfer and PixelMap. We instead used register combiners to shift, randomize and isolate/combine components.

5.1 Register Combiners

Register combiners support very powerful per-pixel operations by combining multitextured lookups in a variety of manners. They support the addition, subtraction and component-wise multiplication (and even a dot product) of RGB vectors. They also support conditional operations based on the high-bit of the alpha channel of one of the inputs. They support signed byte arithmetic with a full 9 bits per channel, though can only store 8 bit results. They also provide several mapping functions for signed/unsigned conversion, and the ability to modulate output values by one-half, two and four.

The Direct3D 8.0 specification includes a register-combiner based assembly language [12]. However, our implementation sought to squeeze the best possible performance out of the NVidia chipset. We chose instead to use the OpenGL register combiner extensions, which provide complete, though device dependent, access to the graphics accelerator.

Figure 10 illustrates the register combiner functionality used in this paper. The register combiner has four inputs A,B,C,D that can be any combination of the incoming fragment, a pixel from multitexture unit 0 or 1, and the contents of a scratch register called Spare0. The constants zero and one (via a special unsigned invert operation) can also be used as inputs, and other constant values can also be loaded via special registers.

The outputs of the register combiners include A*B, C*D, A*B + C*D and the special A*B | C*D. This latter output yields A*B if the alpha component of the register Spare0 is less than 0.5, otherwise the output yields C*D. These outputs can also be optionally scaled by ½, 2 or 4. For this paper, it is safe to assume the output is always contained in the register Spare0. The register combiner has separate but comparable functions for the RGB values and the alpha values of the inputs and registers.

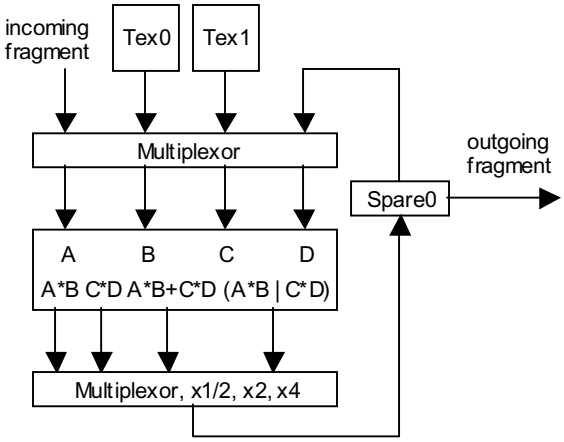


Figure 10. Partial block diagram of the register combiner functionality used in this paper.

There can be any number of register combiners that form a pipeline, using the temporary registers such as Spare0 to hold data between stages. The GeForce2 used to implement the pixel shaders in this paper contains two register combiners which allow two register combiner operations per pass. The GeForce3 is expected to have eight register combiners.

5.2 Logical Shift Operations

In order to perform the decomposition of the input solid texture coordinate image into integer and fractional components, we developed a logical shift left register routine. This routine used the modulate-by-two output mapping, but this causes values greater than one half to clamp to one. We avoided this overflow by using the conditional mode of the register combiners. The following example sets up the register combiners to perform such a logical shift left on a luminance value (R=G=B) in multitexture unit 0.

```
// first stage
// spare[α] = texture0[b]
A[α] = texture0[b]
B[α] = 1 (zero with unsigned_invert)
spare0[α] = A[α]*B[α]
// spare0 rgb = texture0 less its high bit (or zero if less than ½)
A[rgb] = texture0[rgb]
B[rgb] = white (zero with unsigned_invert)
spare0[rgb] = A[rgb]*B[rgb] - 0.5 // via bias_by_negative_one_half

// second stage
// spare0 rgb = (spare0[α] < 0.5 ? texture0[rgb] : spare0[rgb]) << 1
A[rgb] = texture0[rgb]
B[rgb] = white
C[rgb] = spare0[rgb]
D = white
spare0[rgb] = 2*(spare0[α]<0.5 ? A[rgb]*B[rgb] : C[rgb]*D[rgb])
```

We could also generate a register combiner to perform a logical shift right using the scale_by_one_half mode, but found it was much simpler to perform a multitextured modulate-mode blend with a texture consisting of the single pixel containing the RGB color (0.5,0.5,0.5).

5.3 Random Value Synthesis

Randomization on the NVidia controller was particularly difficult. The driver (and presumably the hardware) accelerated neither pixel transfer/mapping operations, nor logical operations like exclusive-or.

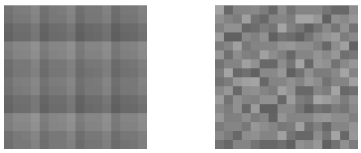
We instead implemented a register combiner random number generator by shifting each of the components of the integer values of the coordinates left one bit at a time. All four bits of each of the three components are at one point the high bit in multitexture unit 0. We then used the register combiner's conditional mode to display one of two colors depending on the high bit of the current texel of multitexture unit 0. The following code fragment implements this technique.

```
for (kk = 0; kk < 4; kk++) {
   for (comp = 0; comp < 3; comp++) {
      // display either tex_ranzero or tex_ranone
      // depending on hi bit of tex_comp
      setupblendhibit(ranzero[comp][kk],ranone[comp][kk]);
      blend2tex(tex_comp[comp],tex_corran);
      savetex(tex_corran);
      if (kk < 3) {
         // shift tex_comp left one
         setupshift1();
         blendtex(tex_comp[comp]);
         savetex(tex_comp[comp]);
      }
   }
}
```

The operation blend2tex(tex_a,tex_b) displays a multitextured image with tex_a as multitexture unit 0 and tex_b as multitexture unit 1.

The arrays ranzero and ranone were initialized with random luminances. These random luminances were used as input to the function setupblendhibit(rgba0,rgba1). This function set up a register combiner that would display either constant color rgba0 or rgba1 depending on the high bit of texture0, and would blend the color (rgba0 or rgba1) with texture1.

We found that setting the alpha channel of rgba0 and rgba1 to 1/8 provided a reasonable balance of colors after twelve successive blending operations. These blends were accumulated in tex_corran (corner random). Note that this loop involves 12 randoms + 9 shifts = 21 passes, which expands to 168 passes for all eight corners.



(a) (b)

Figure 11. Heavily correlated random values generated by blending random colors depending on the bits of the integer lattice value (a). Using (a) to index into a random value reduces the correlation (b).

The resulting tex_corran still exhibited some coordinate correlation, which we reduced with an additional eight single-bit randomizations on tex_corran, yielding tex_corranran. This step resulted in an additional 8 randoms + 7 shifts = 15 passes per corner for a total of 120 passes.

Due to the successive blending, the register combiner noise function is Gaussian distributed. A normal distribution could be recovered through a histogram equalization step, though such operations are not yet accelerated on consumer-level hardware.

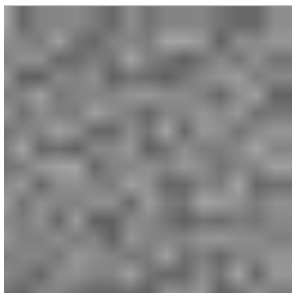


Figure 12. Noise function resulting from register combiners.

5.4 Results

The register combiner implementation resulted in 375 passes, but runs in .77 seconds at a resolution of 256^2 on a GeForce2 using version 12.0 of the "developer" driver. This results in a 1.3 Hz performance, which is suitable for interactive applications but is not yet real-time. A discussion of the reasons why the performance is slower than necessary is given later in Section 6.2.

The resulting noise is shown in Figure 12. The NVidia implementation blended random colors, yielding Gaussian noise, whereas the reference and SGI implementations produced white noise. If desired, one could redistribute the Gaussian noise into white noise with a fixed histogram equalization step, though no such operation is currently accelerated on NVidia GPUs.

6. Discussion

The implementation of the Perlin noise function on SGI and NVidia GPUs has been successful in that we found it was feasible, but disappointing in that subtle hardware limitations prevent truly efficient implementations. These limitations included the limited precision available in the 8 bit per component framebuffer, the delay in performing a CopyTexSubImage transfer from the framebuffer to the texture memory, and the lack of acceleration of loginal operation blend modes such as exclusive-or. The process has also been illuminating, and has inspired us with several ideas for further advancement in hardware design to overcome these limitations and better support efficient multipass pixel shading.

6.1 Limited Precision

Most of the per-pixel operations need only a single channel, and set R=G=B since this is the most efficient mode of operation. The register combiners can be implemented to a higher precision, but their input and output precision is limited to the framebuffer precision.

The register combiners currently support a conversion between 8-bit unsigned external values and 9-bit signed internal values. These conversions perform the function $f(x) = 2x - 1$ on an input, and $f^{-1}(x) = 0.5x + 0.5$ on the output, where x is each of the components of an RGBA pixel.

We could likewise create a packed luminance conversion to the input and output of the register combiners. The input mapping would perform the function L = R << 16 | G << 8 | B yielding a 24-bit luminance value on which one could perform scalar register combiner operations. Internally, the register combiner could maintain a 16.8 fixed-point format, and support operations such as addition, subtraction, multiplication and division using the extended range and precision of the new format. Once the operation is completed, the result may then be unpacked into the 8-bit framebuffer with the output mapping R = L >> 16, G = (L>>8)&0xff and B & 0xff.

6.2 Swizzle-Blits

Given the number of passes required, the register combiner performance was astounding, currently 1.3 Hz on a GeForce2 graphics accelerator at a resolution of 256x256. Profiling the code revealed that the main bottleneck was the time it took to save the framebuffer to a texture, adding an average of 2 ms per pass for 354 of the passes. OpenGL currently does not support rendering directly to texture, and the register combiner does not allow the framebuffer to be used as an input.

Whereas framebuffer memory is organized in scanline order, modern texture memory is organized into blocks and other patterns to better capitalize on spatial coherence. This coherence allows texture pixels to be more effectively cached during texture mapping operation. However, in this case the layout of texture memory is counterproductive. The cost to "swizzle" the memory into the clustered arrangement when saving a framebuffer image to texture memory dominates the execution time of iterative multipass shaders.

We have verifies this delay with a profile of the code, revealing that our CopyTexSubImage operations were taking longer than any other component of our shader. We also experimented with various resolutions and found a direct 1:1 correspondence between the number of pixels and the execution time.

Perhaps a mode can be incorporated into the graphics accelerator state that optionally defeats the spatial-coherent clustering of texture memory. This mode could be enabled during multipass shader evaluation, to eliminate the shuffled memory delay incurred during the CopyTexSubImage operations.

Alternatively, upcoming modes that support rendering directly to texture may also ameliorate this problem.

6.3 Logical Blend Modes

Blending modes such as exclusive-or and logical shifts left and right are extremely valuable when generating random values. Unfortunately these operations are not accelerated under current graphics drivers. Such operations are of the simplest to implement in hardware, and we suspect they will become accelerated as demand for them increases.

7. Conclusion

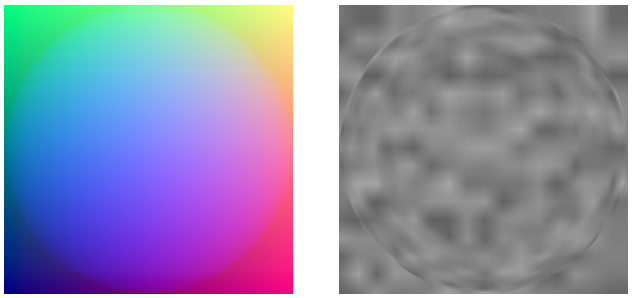
We have investigated the implementation of the Perlin noise function as a multipass pixel shader. We have developed a general algorithm and implemented it using the accelerated features from two different manufacturers.

The SGI implementation based on PixelTransfer and PixelMap operations remains faster than the NVidia implementation based on register combiners. However, we expect the additional register combiner stages available in the upcoming GeForce3 will close this gap.

The process of implementing a general-purpose procedure using GPU accelerated operations has been illuminating. We are excited by the prospect of using the GPU as a SIMD-based supercomputer. However, this vision has been stifled by the low precision available in the buffers and processors, and the latency due to slow framebuffer-to-texture memory transfers. We believe both problems can be solved with moderate changes to existing graphics accelerator architectures, and have suggested possible solution implementations.

Our noise implementation uses linear interpolation of random values on an integer lattice. One can also implement cubic interpolation at the expense of four extra passes. The function SCURVE(u) = $3u^2 - 2u^3$ can also be expressed as $uu(3-2u)$. The function 1/4 SCURVE(u) can be implement by modulating the images u, u and 3/4 – 1/2 u. Note the latter is necessarily scaled by ¼ to fall within the legal [0,1] OpenGL range. This result can then be scaled by 4 (either through PixelTransfer or a register combiner) to yield SCURVE(u).

We have investigated numerous methods for enhancing the performance of these multipass pixel shaders. The 2-D s-t plane examples suggested that image processing applications such as translation and convolution could be applied, but such techniques would not work for arbitrarily shaped objects in the solid texture coordinate image, such as in Figure 13.



(a) (b)

Figure 13. Application of the noise function (b) on a sphere of solid texture coordinates (a).

The source code and an executable for both implementations of the Perlin noise pixel shader can be found at:

http://graphics.cs.uiuc.edu/~jch/mpnoise.zip

Acknowledgments

Conversations with Pat Hanrahan and Henry Moreton were helpful in determining the cause of the 3ms CopyTexSubImage delay. This research was supported in part by a grant from the Evans & Sutherland Computer Corp. Thanks also to Nate Carr for proofreading the paper.

References

[1] Apodaca, A.A. Advanced Renderman: Creating CGI for Motion Pictures. Morgan Jaufmannm 1999. See also: Renderman Tricks Everyone Should Know, in SIGGRAPH 98 or SIGGRAPH 99 Advanced Renderman Course Notes.

[2] Carr, N.A. and J.C. Hart. Real-Time Procedural Solid Texturing. Manuscript, in review. Apr. 2001.

[3] Ebert, D., F.K. Musgrave, D. Peachey, K. Perlin and S. Worley. Texturing and Modeling: A Procedural Approach, Academic Press.1994.

[4] Goehring, D. and O. Gerlitz. Advanced procedural texturing using MMX technology. Intel MMX Technology Application Note, Oct. 1997. http://developer.intel.com/software/idap/resources/technical_collateral/mmx/proctex2.htm

[5] Hanrahan, P. and J. Lawson. A language for shading and lighting calculations. *Computer Graphics* 24(4), (*Proc. SIGGRAPH 90*), Aug. 1990, pp. 289-298.

[6] Hanrahan, P. Procedural shading (keynote). Eurographics / SIGGRAPH Workshop on Graphics Hardware, Aug. 1999. http://graphics.standford.edu/hanrahan/talks/rts1/slides.

[7] Hart, J. C., N. Carr, M. Kameya, S. A. Tibbits, and T.J Colemen. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Aug. 1999, pp. 45-53.

[8] Heidrich, W. and H.-P. Seidel. Realistic hardware-accelerated shading and lighting. *Proc. SIGGRAPH 99*, Aug. 1999, pp. 171-178.

[9] Heidrich, W., R. Westermann, H-P Seidel and T. Ertl. Applications of Pixel Textures in Visualization and Realistic Image Synthesis. Proc. ACM Sym. on Interactive 3D Graphics, Apr. 1999, pp. 127-134.

[10] Kameya, M. and J.C. Hart. Bresenham noise. *SIGGRAPH 2000 Conference Abstracts and Applications*, July 2000.

[11] McCool, M.C. and W. Heidrich. Texture Shaders. *1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Aug. 1999, pp. 117-126.

[12] Microsoft Corp. Direct3D 8.0 specification. Available at: http://www.msdn.microsoft.com/directx.

[13] Mine, A. and F. Neyret. Perlin Textures in Real Time using OpenGL. Research Report #3713, INRIA, 1999. http://www-imagis.imag.fr/Membres/Fabrice.Neyret/publis/RR-3713-eng.html

[14] NVidia Corp. Noise, component of the NVEffectsBrowser. Available at: http://www.nvidia.com/developer.

[15] Olano, M. and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. Proc. SIGGRAPH 98, July 1998, pp. 159-168.

[16] OpenGL Architecture Review Board. OpenGL Extension Registry. Available at: http://oss.sgi.com/projects/ogl-sample/registry/

[17] Peachey, D.R. Solid texturing of complex surfaces. *Computer Graphics* 19(3), July 1985, pp. 279-286.

[18] Peercy, M.S., M. Olano, J. Airey and P.J. Ungar. Interactive multi-pass programmable shading, Proc. SIGGRAPH 2000, July 2000, pp. 425-432.

[19] Perlin, K. An image synthesizer. *Computer Graphics* 19(3). July 1985, pp. 287-296.

[20] Pixar Animation Studios. Future requirements for graphics hardware. Memo, 12 April 1999.

[21] Proudfoot, K., W.R. Mark, S. Tzvetkov and P. Hanrahan. A real-time programmable shading system for programmable graphics hardware. Proc. SIGGRAPH 2001, Aug. 2001, to appear.

[22] Rhoades, J., G. Turk, A. Bell, U. Neumann, and A. Varshney. Real-time procedural textures. 1992 *Symposium on Interactive 3D Graphics* 25(2), March 1992, pp 95-100.

[23] Segal, M. and K. Akeley. The OpenGL Graphics System: A Specification, Version 1.2.1. Available at: http://www.opengl.org/.

[24] Skinner, R. and C.E. Kolb. noise.c component of the Rayshade ray tracer, 1991.

Chapter 7

Shading Through Multi-Pass Rendering

Marc Olano

Interactive Multi-Pass Programmable Shading

Mark S. Peercy, Marc Olano, John Airey*, P. Jeffrey Ungar
SGI

Abstract

Programmable shading is a common technique for production animation, but interactive programmable shading is not yet widely available. We support interactive programmable shading on virtually any 3D graphics hardware using a scene graph library on top of OpenGL. We treat the OpenGL architecture as a general SIMD computer, and translate the high-level shading description into OpenGL rendering passes. While our system uses OpenGL, the techniques described are applicable to any retained mode interface with appropriate extension mechanisms and hardware API with provisions for recirculating data through the graphics pipeline.

We present two demonstrations of the method. The first is a constrained shading language that runs on graphics hardware supporting OpenGL 1.2 with a subset of the ARB imaging extensions. We remove the shading language constraints by minimally extending OpenGL. The key extensions are *color range* (supporting extended range and precision data types) and *pixel texture* (using framebuffer values as indices into texture maps). Our second demonstration is a renderer supporting the RenderMan Interface and RenderMan Shading Language on a software implementation of this extended OpenGL. For both languages, our compiler technology can take advantage of extensions and performance characteristics unique to any particular graphics hardware.

CR categories and subject descriptors: I.3.3 [Computer Graphics]: Picture/Image generation; I.3.7 [Image Processing]: Enhancement.

Keywords: Graphics Hardware, Graphics Systems, Illumination, Languages, Rendering, Interactive Rendering, Non-Realistic Rendering, Multi-Pass Rendering, Programmable Shading, Procedural Shading, Texture Synthesis, Texture Mapping, OpenGL.

1 INTRODUCTION

Programmable shading is a means for specifying the appearance of objects in a synthetic scene. Programs in a special purpose language, known as *shaders*, describe light source position and emission characteristics, color and reflective properties of surfaces, or transmittance properties of atmospheric media. Conceptually, these programs are executed for each point on an object as it is being rendered to produce a final color (and perhaps opacity) as seen from a given viewpoint. Shading languages can be quite general, having constructs familiar from general purpose programming languages such as `C`, including loops, conditionals, and functions. The most common is the RenderMan Shading Language [32].

The power of shading languages for describing intricate lighting and shading computations been widely recognized since Cook's seminal shade tree research [7]. Programmable shading has played a fundamental role in digital content creation for motion pictures and television for over a decade. The high level of abstraction in programmable shading enables artists, storytellers, and their technical collaborators to translate their creative visions into images more easily. Shading languages are also used for visualization of scientific data. Special *data shaders* have been developed to support the depiction of volume data [3, 8], and a texture synthesis language has been used for visualizing data fields on surfaces [9]. Image processing scripting languages [22, 31] also share much in common with programmable shading.

Despite its proven usefulness in software rendering, hardware acceleration of programmable shading has remained elusive. Most hardware supports a parametric appearance model, such as Phong lighting evaluated per vertex, with one or more texture maps applied after Gouraud interpolation of the lighting results [29]. The general computational nature of programmable shading, and the unbounded complexity of shaders, has kept it from being supported widely in hardware. This paper describes a methodology to support programmable shading in interactive visual computing by compiling a shader into multiple passes through graphics hardware. We demonstrate its use on current systems with a constrained shading language, and we show how to support general shading languages with only two hardware extensions.

1.1 Related Work

Interactive programmable shading, with dynamically changing shader and scene, was demonstrated on the PixelFlow system [26]. PixelFlow has an array of general purpose processors that can execute arbitrary code at every pixel. Shaders written in a language based on RenderMan's are translated into `C++` programs with embedded machine code directives for the pixel processors. An application accesses shaders through a programmable interface extension to OpenGL. The primary disadvantages of this approach are the additional burden it places on the graphics hardware and driver software. Every system that supports a built-in programmable interface must include powerful enough general computing units to execute the programmable shaders. Limitations to these computing units, such as a fixed local memory, will either limit the shaders that may be run, have a severe impact on performance, or cause the system to revert to multiple passes within the driver. Further, every such system will have a unique shading language compiler as part of the driver software. This is a sophisticated piece of software which greatly increases the complexity of the driver.

Our approach to programmable shading stands in contrast to the programmable hardware method. Its inspiration is a long line of interactive algorithms that follow a general theme: treat the graphics hardware as a collection of primitive operations that can be used

*Now at Intrinsic Graphics

to build up a final solution in multiple passes. Early examples of this model include multi-pass shadows, planar reflections, highlights on top of texture, depth of field, and light maps [2, 10]. There has been a dramatic surge of research in this area over the past few years. Sophisticated appearance computations, which had previously been available only in software renderers, have been mapped to generic graphics hardware. For example, lighting per pixel, general bidirectional reflectance distribution functions, and bump mapping now run in real-time on hardware that supports none of those effects natively [6, 17, 20, 24].

Consumer games like ID Software's Quake 3 make extensive use of multi-pass effects [19]. Quake 3 recognizes that multi-pass provides a flexible method for surface design and takes the important step of providing a scripting mechanism for rendering passes, including control of OpenGL blending mode, alpha test functions, and vertex texture coordinate assignment. In its current form, this scripting language does not provide access to all of the OpenGL state necessary to treat OpenGL as a general SIMD machine.

A team at Stanford has been investigating real-time programmable shading. Their focus is a framework and language that explicitly divides operations into those that are executed at the vertex processing stage in the graphics pipeline and those that are executed at the fragment processing stage [25].

The hardware in all of these cases is being used as a computing machine rather than a special purpose accelerator. Indeed, graphics hardware has been used to accelerate techniques such as back-projection for tomographic reconstruction [5] and radiosity approximations [21]. It is now recognized that some new hardware features, such as multi-texture [24, 29], pixel texture [17], and color matrix [23], are particularly valuable for supporting these advanced computations interactively.

1.2 Our Contribution

In this paper, we embrace and extend previous multi-pass techniques. We treat the OpenGL architecture as a SIMD computer. OpenGL acts as an assembly language for shader execution. The challenge, then, is to convert a shader into an efficient set of OpenGL rendering passes on a given system. We introduce a compiler between the application and the graphics library that can target shaders to different hardware implementations.

This philosophy of placing the shading compiler above the graphics API is at the core of our work, and has a number of advantages. We believe the number of languages for interactive programmable shading will grow and evolve over the next several years, responding to the unique performance and feature demands of different application areas. Likewise, hardware will increase in performance and many new features will be introduced. Our methodology allows the languages, compiler, and hardware to evolve independently because they are cleanly decoupled.

This paper has three main contributions. First, we formalize the idea of using OpenGL as an assembly language into which programmable shaders are translated, and we show how to apply dynamic tree-rewriting compiler technology to optimize the mapping between shading languages and OpenGL (Section 2). Second, we demonstrate the immediate application of this approach by introducing a constrained shading language that runs interactively on most current hardware systems (Section 3). Third, we describe the color range and pixel texture OpenGL extensions that are necessary and sufficient to accelerate fully general shading languages (Section 4). As a demonstration of the viability of this solution, we present a complete RenderMan renderer including full support of the RenderMan Shading Language running on a software implementation of this extended OpenGL. We close the paper with a discussion (Section 5) and conclusion (Section 6).

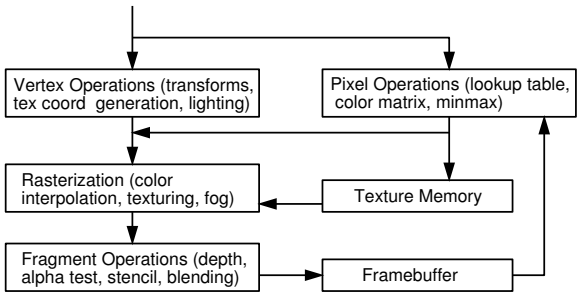


Figure 1: A simplified block diagram of the OpenGL architecture. Geometric data passes through the vertex operations, rasterization, and fragment operations to the framebuffer. Pixel data (either from the host or the framebuffer) passes through the pixel operations and on to either texture memory or through the fragment pipeline to the framebuffer.

2 THE SHADING FRAMEWORK

There is great diversity in modern 3D graphics hardware. Each graphics system includes unique features and performance characteristics. Countering this diversity, all modern graphics hardware also supports the basic features of the OpenGL API standard.

While it is possible to add shading extensions to graphics hardware, OpenGL is powerful enough to support shading with no extensions at all. Building programmable shading on top of standard OpenGL decouples the hardware and drivers from the language, and enables shading on every existing and future OpenGL-based graphics system.

A compiler turns shading computations into multiple passes through the OpenGL rendering pipeline (Figure 1). This compiler can produce a general set of rendering passes, or it can use knowledge of the target hardware to pick an optimized set of passes.

2.1 OpenGL as an Assembly Language

One key observation allows shaders to be translated into multi-pass OpenGL: a single rendering pass is also a general SIMD instruction — the same operations are performed simultaneously for all pixels in an object. At the simplest level, the framebuffer is an accumulator, texture or pixel buffers serve as per-pixel memory storage, blending provides basic arithmetic operations, lookup tables support function evaluation, the alpha test provides a variety of conditionals, and the stencil buffer allows pixel-level conditional execution. A shader computation is broken into pieces, each of which can be evaluated by an OpenGL rendering pass. In this way, we build up a final result for all pixels in an object (Figure 2). There are typically several ways to map shading operations into OpenGL. We have implemented the following:

Data Types: Data with the same value for every pixel in an object are called *uniform*, while data with values that may vary from pixel to pixel are called *varying*. Uniform data types are handled outside the graphics pipeline. The framebuffer retains intermediate varying results. Its four channels may hold one quadruple (such as a homogeneous point), one triple (such as a vector, normal, point, or color) and one scalar, or four independent scalars. We have made no attempt to handle varying data types with more than four channels. The framebuffer channels (and hence independent scalars or

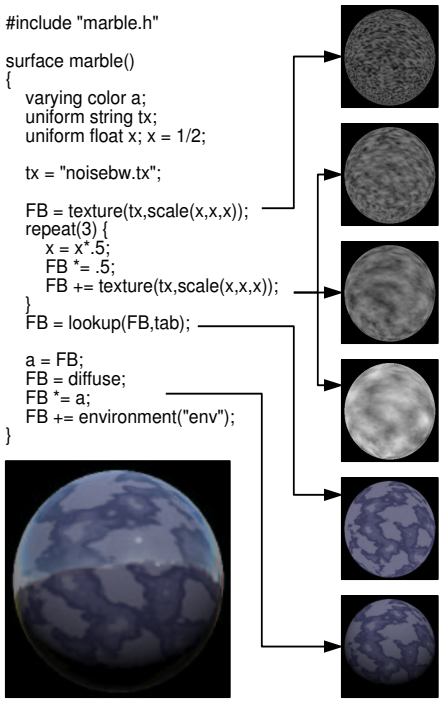


Figure 2: SIMD Computation of a Shader. Some of the different passes for the shader written in ISL listed on the left are shown as thumbnails down the right column. The result of the complete shader is shown on the lower left.

the components of triples and quadruples) can be updated selectively on each pass by setting the write-mask with `glColorMask`.

Variables: Varying global, local, and temporary variables are transferred from the framebuffer to a named texture using `glCopyTexSubImage2D`, which copies a portion of the framebuffer into a portion of a texture. In our system, these textures can be one channel (intensity) or four channels (RGBA), depending on the data type they hold. Variables are used either by drawing a textured copy of the object bounding box or by drawing the object geometry using a projective texture. The relative speed of these two methods will vary from graphics system to graphics system. Intensity textures holding scalar variables are expanded into all four channels during rasterization and can therefore be restored into any framebuffer channel.

Arithmetic Operations: Most arithmetic operations are performed with framebuffer blending. They have two operands: the framebuffer contents and an incoming fragment. The incoming fragment may be produced either by drawing geometry (object color, a texture, a stored variable, etc.) or by copying pixels from the framebuffer and through the pixel operations with `glCopyPixels`. Data can be permuted (*swizzled*) from one framebuffer channel to another or linearly combined more generally using the color matrix during a copy. The framebuffer blending mode, set by `glBlendEquation`, `glBlendFunc`, and `glLogicOp`, supports overwriting, addition, subtraction, multiplication, bit-wise logical operations, and alpha blending. Unextended OpenGL does not have a divide blend mode. We handle divide using multiplication by the reciprocal. The reciprocal is computed like other mathematical functions (see below). More complicated binary operations are reduced to a combination of these primitive operations. For example, a dot product of two vectors is a component-wise multiplication followed by a pixel copy with a color matrix that sums the resulting three components together.

Mathematical and Shader Functions: Mathematical functions with a single scalar operand (e.g. sin or reciprocal) use color or texture lookup tables during a framebuffer-to-framebuffer pixel copy. Functions with more than one operand (e.g. atan2) or a single vector operand (e.g. normalize or color space conversion) are broken down into simpler monadic functions and arithmetic operations, each of which can be supported in a pass through the OpenGL pipeline. Some shader functions, such as texturing and diffuse or specular lighting, have direct correspondents in OpenGL. Often, complex mathematical and shader functions are simply translated to a series of simpler shading language functions.

Flow Control: Stenciling, set by `glStencilFunc` and `glStencilOp`, limits the effect of all operations to only a subset of the pixels, with other pixels retaining their original framebuffer values. We use one bit of the stencil to identify pixels in the object, and additional stencil bits to identify subsets of those pixels that pass varying conditionals (*if-then-else* constructs and loops). One stencil bit is devoted to each level of nesting. Loops with uniform control and conditionals with uniform relations do not need a stencil bit to control their influence because they affect all pixels.

A two step process is used to set the stencil bit for a varying conditional. First, the relation is computed with normal arithmetic operations, such that the result ends up in the alpha channel of the framebuffer. The value is zero where the condition is true and one where it is false. Next, a pixel copy is performed with the alpha $> .5$ test enabled (set by `glAlphaFunc`). Only fragments that pass the alpha test are passed on to the stenciling stage of the OpenGL pipeline. A stencil bit is set for all of these fragments. The stencil remains unchanged for fragments that failed the alpha test. In some cases, the first operation in the body of the conditional can occur in the same pass that sets the stencil.

The passes corresponding to the different blocks of shader code at different nesting levels affect only those pixels that have the proper stencil mask. Because we are executing a SIMD computation, it is necessary to evaluate both branches of *if-then-else* constructs whose relation varies across an object. The stencil compare for the *else* clause simply uses the complement of the stencil bit for the *then* clause. Similarly, it is necessary to repeat a loop with a varying termination condition until all pixels within the object exit the loop. This requires a test that examines all of the pixels within the object. We use the *minmax* function from the ARB imaging extension as we copy the alpha channel to determine if any alpha values are non-zero (signifying they still pass the looping condition). If so, the loop continues.

2.2 OpenGL Encapsulation

We encapsulate OpenGL instructions in three kinds of rendering passes: *GeomPasses*, *CopyPasses*, and *CopyTexPasses*. GeomPasses draw geometry to use vertex, rasterization, and fragment operations. CopyPasses copy a subregion of the framebuffer (via `glCopyPixels`) back into the same place in the framebuffer to use pixel, rasterization, and fragment operations. A stencil allows the CopyPass to avoid operating on pixels outside the object. CopyTexPasses copy a subregion of the framebuffer into a texture object (via `glCopyTexSubImage2D`) and also utilize pixel operations. There are two subtypes of GeomPass. The first draws the object geometry, including normal vectors and texture coordinates. The second draws a screen-aligned bounding rectangle that covers the object using stenciling to limit the operations to pixels on the object. Each pass maintains the relevant OpenGL state for its path

through the pipeline. State changes on drawing are minimized by only setting the state in each pass that is not default and immediately restoring that state after the pass.

2.3 Compiling to OpenGL

The key to supporting interactive programmable shading is a compiler that translates the shading language into OpenGL assembly. This is a CISC-like compiler problem because OpenGL passes are complex instructions. The problem is somewhat simplified due to constraints in the language and in OpenGL as an instruction set. For example, we do not have to worry about instruction scheduling since there is no overlap between rendering passes.

Our compiler implementation is guided by a desire to retarget the compiler to easily take advantage of unique features and performance and to pick the best set of passes for each target architecture. We also want to be able to support multiple shading languages and adapt as languages evolve. To help meet these goals, we built our compiler using an in-house tool inspired by the iburg code generation tool [11], though we use it for all phases of compilation. This tool finds the least-cost covering of a tree representation of the shader based on a text file of patterns.

A simple example can show how the tree-matching tool operates and how it allows us to take advantage of extensions to OpenGL. Part of a shader might be matched by a pair of texture lookups, each with a cost of one, or by a single multi-texture lookup, also with a cost of one. In this case, multi-texture is cheaper because it has a total cost of one instead of two. Using similar matching rules and semantic actions, the compiler can make use of fragment lighting, light texture, noise generation, divide or conditional blends, or any other OpenGL extension [16, 27].

The entire shader is matched at once, giving the set of matching rules that cover the shader with the least total cost. For example, the computations surrounding the above pair of texture lookups expand the set of possible matching rules. Given operation A, texture lookup B, texture lookup C, and operation D, it may be possible to do all of the operations in four separate passes (A,B,C,D), to do the surrounding operations separately while combining the texture lookups into one multi-texture pass for a total cost of three (A,BC,D), or to combine one computation with each texture lookup for a cost of two (AB,CD). By considering the entire shader we can choose the set of matching rules with the least overall cost.

When we use the tool for final OpenGL pass generation, we currently use the number of passes as the cost for each matching rule. For performance optimization, the costs should correspond to predicted rendering speed, so the cost for a GeomPass would be different from the cost for a CopyPass or a CopyTexPass.

The pattern matching happens in two phases, *labeling* and *reducing*. Labeling is done bottom-up through the abstract syntax tree, using dynamic programming to find the least-cost set of pattern match rules. Reducing is done top-down, with one semantic action run before the node's children are reduced and one after. The iburg-like label/reduce tool proved useful for more than just final pass selection. We use it for shader syntax checking, constant folding, and even memory allocation (although most of the memory allocation algorithm is in the code associated with a small number of rules). The ease of changing costs and creating new matching rules allows us to achieve our goal of flexible retargeting of the compiler for different hardware and shading languages.

2.4 Scene Graph Support

Since objects may be rendered multiple times, it is necessary to retain geometry data and to deliver it repeatedly to the graphics hardware. In addition, shaders need to be associated with objects to describe their appearances, and the shaders and objects need to be translated into OpenGL passes to render an image. Our framework supports these operations in a scene graph used by an application through the addition of new scene graph containers and new traversals.

In our implementation, we have extended the Cosmo3D scene graph library [30]. Cosmo3D uses a familiar hierarchical scene graph. Internal nodes describe coordinate transformations, while the leaves are *Shape* nodes, each of which contains a list of *Geometry* and an *Appearance*. Traversals of the scene graph are known as *actions*. A *DrawAction*, for example, is applied to the scene graph to render the objects into a window.

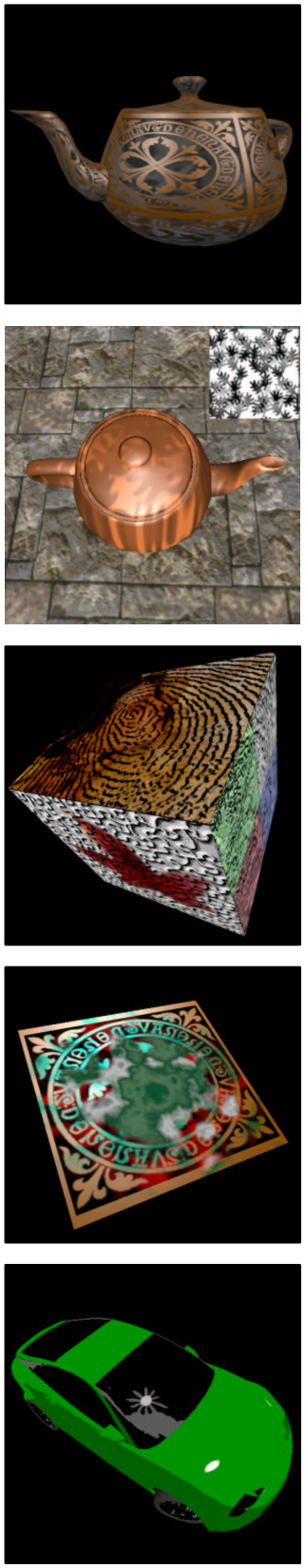
We have implemented a new appearance class that contains shaders. When included in a shape node, this appearance completely describes how to shade the geometry in the shape. The shaders may include a list of active light shaders, a displacement shader, a surface shader, and an atmosphere shader. In addition, we have implemented a new traversal, known as a *ShadeAction*. A ShadeAction converts a scene graph containing shapes with the new appearance into another Cosmo3D scene graph describing the multiple passes for all of the objects in the original scene graph. (The transformation of scene graphs is a powerful, general technique that has been proposed to address a variety of problems [1].) The key element of the ShadeAction is the shading language compiler that converts the shaders into multiple passes. A ShadeAction may treat multiple objects that share the same shader as a single, combined object to minimize overhead. A DrawAction applied to this second scene graph renders the final image.

The scene graph passes information to the compiler including the matrix to transform from the object's coordinate system into camera space and the screen space footprint for the geometry. The footprint is computed during the ShadeAction by projecting a 3D bounding box of the geometry into screen space and computing an axis-aligned 2D bounding box of the eight projected points. Only pixels within the 2D bounding box are copied on a CopyPass or drawn on the quad-GeomPass to minimize unnecessary data movement when shading each object.

We provide support for debugging at the single-step, pass-by-pass level through special hooks inserted into the DrawAction. Each pass is held in an extended Cosmo3D *Group* node, which invokes the debugging hook functions when drawn. Each pass is also tagged with the line of source code that generated it, so everything from shader source-level debugging to pass-by-pass image dumps is possible. Hooks at the per-pass level also let us monitor or estimate performance. At the coarsest level, we can find the number of passes executed, but we can also examine each pass to record details like pixels written or time to draw.

3 EXAMPLE: INTERACTIVE SL

We have developed a constrained shading language, called ISL (for Interactive Shading Language) [25] and an ISL compiler to demonstrate our method on current hardware. ISL is similar in spirit to the RenderMan Shading Language in that it provides a `C`-like syntax to specify per-pixel shading calculations, and it supports separate light, surface, and atmosphere shaders. Data types include varying colors, and uniform floats, colors, matrices, and strings. Local variables can hold both uniform and varying values. Nestable flow control structures include loops with uniform control, and uniform and varying conditionals. There are built-in functions for diffuse and specular lighting, texture mapping, projective textures, environment mapping, RGBA one-dimensional lookup tables, and per-pixel ma-



```
surface celtic() {
    varying color a;
    FB = diffuse;
    FB *= color(.5,.2,0.,1.);
    a = FB;
    FB = specular(30.);
    FB += a;
    FB *= texture("celtic");
    a = FB;
    FB = 1;
    FB −= texture("celtic");
    FB *= texture("silk");
    FB *= .15;
    FB += a;
}
distantlight leaves(uniform string
                map = "leaves", ...) {
    uniform float tx;
    uniform float ty;
    uniform float tz;
    tx = frame*speedx+phasex;
    ty = frame*speedy+phasey;
    tz = frame*speedz+phasez;
    FB = project(map,
            scale(sx,sx,sx)*
            rotate(0,0,1,rx)*
            translate(ax*sin(tx),0,0)*
            shadermatrix);
    FB *= project(map,
            scale(sy,sy,sy)*...);
}
uniform matrix lt = (0,0,0,0,
     0,0,0,0,1,1,1,0,0,0,0,1);
surface bump(uniform string b="";
      uniform string tx =  "") {
    uniform matrix m;
    FB = texture(b);
    m = objectmatrix;
    m[0][3] = m[1][3] = m[2][3] = 0.;
    m[3][3] = m[3][0] = m[3][1] = 0.;
    m[3][2] = 0.;
    m = lt*m*translate(−1,−1,−1)*
            scale(2,2,2);
    FB = transform(FB,m);
    FB *= texture(tx);
}
#include "threshtab.h"
surface shipRockRot(...) {
    varying color a, b, c;
    FB = texture(rot); FB *= .5;
    FB += .32*(1−cos(.08*frame));
    FB = lookup(FB,mtab); c = FB;
    FB = color(1,1,1,1); FB −= c;
    FB *= texture(t1); a = FB;
    FB = texture(t2);
    FB *= texture(rot);
    FB = diffuse;
    FB *= color(.5,.2,0,1); b = FB;
    FB = specular(30.);
    FB += b; FB *= texture(t2);
    FB *= c; FB += a;
}

#include "swizzle.h"
table greentable = { {0,.2,0,1},
                     {0,.4,0,1) };
surface toon(uniform float do = 1.;
     uniform float edge = .25 ) {
    FB = environment("park.env");
    if (do > .5) {
        FB += edge;
        FB =transform(FB,rgba_rrra);
        FB =lookup(FB,greentable);
        FB += environment("sun");
    }
}
```

Figure 3: ISL Examples. ISL shaders are shown to the right of each image. Ellipses denote where parameters and statements have been omitted. Some tables are in header files.

trix transformations. In addition, ISL supports uniform shader parameters and a set of uniform global variables (shader space, object space, time, and frame count).

We have intentionally constrained ISL in a number of ways. First, we only chose primitive operations and built-in functions that can be executed on any hardware supporting base OpenGL 1.2 plus the color matrix extension. Consequently, many current hardware systems can support ISL. (If the color matrix transformation is eliminated, ISL should run anywhere.) This constraint provides the shader writer with insight into how limited precision of current commercial hardware may affect the shader. Second, the syntax does not allow varying expressions of expressions, which ensures that the compiler does not need to create any temporary storage not already made explicit in the shader. As a result, the writer of a shader knows by inspection the worst-case temporary storage required by the shading code (although the compiler is free to use less storage, if possible). Third, arbitrary texture coordinate computation is not supported. Texture coordinates must come either from the geometry or from the standard OpenGL texture coordinate generation methods and texture matrix.

One consequence of these design constraints is that ISL shading code is largely decoupled from geometry. For example, since shader parameters are uniform there is no need to attach them directly to each surface description in the scene graph. As a result, ISL and the compiler can migrate from application to application and scene graph to scene graph with relative ease.

3.1 Compiler

We perform some simple optimizations in the parser. For instance, we do limited constant compression by evaluating at parse time all expressions that are declared uniform. When parameters or the shader code change, we must reparse the shader. In our current system, we do this every time we perform a ShadeAction. A more sophisticated compiler, such as the one implemented for the RenderMan Shading Language (Section 4) performs these optimizations outside the parser.

We expand the parse trees for all of the shaders in an appearance (light shaders, surface shader, and atmosphere shader) into a single tree. This tree is then labeled and reduced using the tree matching compiler tool described in Section 2.3. The costs fed into the labeler instruct the compiler to minimize the total number of passes, regardless of the relative performance of the different kinds of passes.

The compiler recognizes and optimizes subexpressions such as a texture, diffuse, or specular lighting multiplied by a constant. The compiler also recognizes when a local variable is assigned a value that can be executed in a single pass. Rather than executing the pass, storing the result, and retrieving it when referenced, the compiler simply replaces the local variable usage with the single pass that describes it.

3.2 Demonstration

We have implemented a simple viewer on top of the extended scene graph to demonstrate ISL running interactively. The viewer supports mouse interaction for rotation and translation. Users can also modify shaders interactively in two ways. They can edit shader text files, and their changes are picked up immediately in the viewer. Additionally, they can modify parameters by dragging sliders, rotating thumb-wheels, or entering text in a control panel. The viewer creates the control panel on the fly for any selected shader. Changes to the parameters are seen immediately in the window. Examples of the viewer running ISL are given in Figures 2 and 3.

4 EXAMPLE: RENDERMAN SL

RenderMan is a rendering and scene description interface standard developed in the late 1980s [14, 28, 32]. The RenderMan standard includes procedural and bytestream scene description interfaces. It also defines the RenderMan Shading Language, which is the *de facto* standard for programmable shading capability and represents a well-defined goal for anyone attempting to accelerate programmable shading.

The RenderMan Shading Language is extremely general, with control structures common to many programming languages, rich data types, and an extensive set of built-in operators and geometric, mathematical, lighting, and communication functions. The language originally was designed with hardware acceleration in mind, so complicated or user-defined data types that would make acceleration more difficult are not included. It is a large but straightforward task to translate the RenderMan Shading Language into multi-pass OpenGL, assuming the following two extensions:

Extended Range and Precision Data Types: Even the simplest RenderMan shaders have intermediate computations that require data values to extend beyond the range [0-1], to which OpenGL fragment color values are clamped. In addition, they need higher precision than is found in current commercial hardware. With the *color range* extension, color data can have an implementation-specific range to which it is clamped during rasterization and framebuffer operations (including color interpolation, texture mapping, and blending). The framebuffer holds colors of the new type, and the conversion to a displayable value happens only upon video scan-out. We have used the color range extension with an IEEE single precision floating point data type or a subset thereof to support the RenderMan Shading Language.

Pixel Texture: RenderMan allows texture coordinates to be computed procedurally. In this case, texture coordinates cannot be expected to change linearly across a geometric primitive, as required in unextended OpenGL. This general two-dimensional indirection mechanism can be supported with the OpenGL pixel texture extension [17, 18, 27]. This extension allows the (possibly floating point) contents of the framebuffer to be used as texture indices when pixels are copied from the framebuffer. The red, green, blue, and alpha channels are used as texture coordinates s, t, r, and q, respectively. We use pixel texture not only to index two dimensional textures but also to index extremely wide one-dimensional textures. These wide textures are used as lookup tables for mathematical functions such as sin, reciprocal, and sqrt. These can be simple piecewise linear approximations, starting points for Newton iteration, components used to construct the more complex mathematical functions, or even direct one-to-one mappings for a reduced floating point format.

4.1 Scene Graph Support

The RenderMan Shading Language demands greater support from the scene graph library than ISL because geometry and shaders are more tightly coupled. *Varying parameters* can be supplied as four values that correspond to the corners of a surface patch, and the parameter over the surface is obtained through bilinear interpolation. Alternatively, one parameter value may be supplied per control point for a bicubic patch mesh or a NURBS patch, and the parameter is interpolated using the same basis functions that define the surface. We associate a (possibly empty) list of named parameters with each surface to hold any parameters provided when the surface is defined. When the surface geometry is tessellated to form *GeoSets* (triangle strip sets and fan sets, etc.), its parameters are transferred to the GeoSets so that they may be referenced

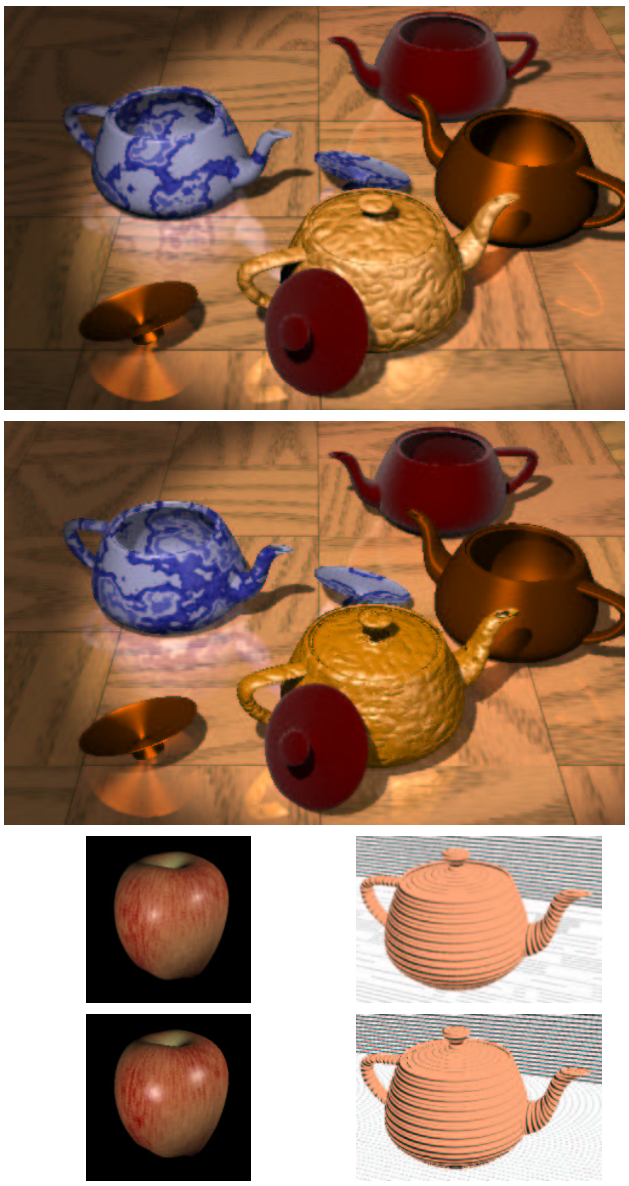


Figure 4: RenderMan SL Examples. The top and bottom images of each pair were rendered with PhotoRealistic RenderMan from Pixar and our multi-pass OpenGL renderer, respectively. No shaders use image maps, except for the reflection and depth shadow maps generated on the fly. The wood floor, blue marble, red apple, and wood block print textures all are generated procedurally. The velvet and brushed metal shaders use sophisticated `illuminance` blocks for their reflective properties. The specular highlight differences are due to Pixar's proprietary specular function; we use the definition from the RenderMan specification. The blue marble, wood floor, and apple do not match because of differences in the `noise` function. Other discrepancies typically are due to limited precision lookup tables used to help evaluate mathematical functions. (Credit: LGParquetPlank by Larry Gritz, SHWvelvet and SHWbrushedmetal by Stephen Westin, DPBlueMarble by Darwin Peachey, eroded from the RenderMan companion, JMredapple by Jonathan Merritt, and woodblockprint by Scott Johnston. Courtesy of the RenderMan Repository `http://www.renderman.org`.)

and drawn as vertex colors by the passes produced by the compiler. Similarly, a shader may require derivatives of surface properties, such as the partial derivatives of the position (*dP/du* and *dP/dv*) either as global variables or through a differential function such as `calculatenormal`. A shader may also use derivatives of user-supplied parameters. The compiler can request from the scene graph any of these quantities evaluated over a surface at the same points used in its tessellation. As with any other parameter, they are computed on the host and stored in the vertex colors for the surface. Where possible, lazy evaluation ensures that the user does not pay in time or space for this support unless requested.

4.2 Compiler

Our RenderMan compiler is based on multiple phases of the tree-matching tool described in Section 2.3. The phases include:

Parsing: convert source into an internal tree representation.
Phase0: detect errors
Phase1: perform context-sensitive typing (e.g. noise, texture)
Phase2: detect and compress uniform expressions
Phase3: compute "difference trees" for Derivatives
Phase4: determine variable usage and live range information
Phase5: identify possible OpenGL instruction optimizations
Phase6: allocate memory for variables
Phase7: generate optimized, machine specific OpenGL

The mapping of RenderMan to OpenGL follows the methodology described in Section 2.1. Texturing and some lighting carry over directly; most math functions are implemented with lookup tables; coordinate transformations are implemented with the color matrix; loops with varying termination condition are supported with minmax; and many built-in functions (including illuminance, solar, and illuminate) are rewritten in terms of simpler operations. Features whose mapping to OpenGL is more sophisticated include:

Noise: The RenderMan SL provides band-limited `noise` primitives that include 1D, 2D, 3D, and 4D operands and single or multiple component output. We use floating point arithmetic and texture tables to support all of these functions.

Derivatives: The RenderMan SL provides access to surface-derivative information through functions that include `Du`, `Dv`, `Deriv`, `area`, and `calculatenormal`. We dedicate a compiler phase to fully implement these functions using a technique similar that described by Larry Gritz [12].

A number of optimizations are supported by the compiler. Uniform expressions are identified and computed once for all pixels. If texture coordinates are linear functions of *s* and *t* or vertex coordinates, they are recognized as a single pass with some combination of texture coordinate generation and texture matrix. Texture memory utilization is minimized by allocating storage based on single-static assignment and live-range analysis [4].

4.3 Demonstration

We have implemented a RenderMan renderer, complete with shading language, bytestream, and procedural interfaces on a software implementation of OpenGL including color range and pixel texture. We experimented with subsets of IEEE single precision floating point. An interesting example was a 16 bit floating point format with a sign bit, 10 bits of mantissa and 5 bits of exponent. This format was sufficient for most shaders, but fell short when computing derivatives and related difference-oriented functions such as `calculatenormal`. Our software implementation supported other OpenGL extensions (cube environment mapping, fragment lighting, light texture, and shadow), but they are not strictly necessary as they can all be computed using existing features.

ISL Image	celtic	leaves	bump	rot	toon
MPix Filled	2.8	4.3	1.2	2.2	1.9
Frames/Second	6.8	7.3	9.6	12.5	4.6
RSL Image	teapots	apple	print		
MPix Filled	500	280	144		

Table 1: Performance for 512x512 images on Silicon Graphics Octane/MXI

The RenderMan bytestream interface was implemented on top of the RenderMan procedural interface. When data is passed to the procedural interface, it is incorporated into a scene graph. Higher order geometric primitives not native to Cosmo3D, such as trimmed quadrics and NURBS patches are accommodated by extending the scene graph library with parametric surface types, which are tessellated just before drawing. At the WorldEnd procedural call, this scene graph is rendered using a ShadeAction that invokes the RenderMan shading language compiler followed by a DrawAction.

To establish that the implementation was correct, over 2000 shading language tests, including point-feature tests, publicly available shaders, and more sophisticated shaders were written or obtained. The results of our renderer were compared to Pixar's commercially available PhotoRealistic RenderMan renderer. While never bit-for-bit accurate, the shading is typically comparable to the eye (with expected differences due, for instance, to the `noise` function). A collection of examples is given in Figure 4. We focused primarily on the challenge of mapping the entire language to OpenGL, so there is considerable room for further optimization.

There are a few notable limitations in our implementation. Displacement shaders are implemented, but treated as bump mapping shaders; surface positions are altered only for the calculation of normals, not for rasterization. True displacement would have to happen during object tessellation and would have performance similar to displacement mapping in traditional software implementations. Transparency is not implemented. It is possible, but requires the scene graph to depth-sort potentially transparent surfaces. Pixel texture, as it is implemented, does not support texture filtering, which can lead to aliasing. Our renderer also does not currently support high quality pixel antialiasing, motion blur, and depth of field. One could implement all of these through the accumulation buffer as has been demonstrated elsewhere [13].

5 DISCUSSION

We measured the performance of several of our ISL and RenderMan shaders (Table 1). The performance numbers for millions of pixels filled are conservative estimates since we counted all pixels in the object's 2D bounding box even when drawing object geometry that touched fewer pixels.

5.1 Drawbacks

Our current system has a number of inefficiencies that impact our performance. First, since we do not use deferred shading, we may spend several passes rendering an object that is hidden in the final image. There are a variety of algorithms that would help (for example, visibility culling at the scene graph level), but we have not implemented any of them.

Second, the bounding box of objects in screen space is used to define the active pixels for many passes. Consequently pixels within the bounding box but not within the object are moved unnecessarily. This taxes one of the most important resources in hardware: bandwidth to and from memory.

Third, we have only included a minimal set of optimization rules in our compiler. Many current hardware systems share framebuffer and texture memory bandwidth. On these systems, storage and retrieval of intermediate results bears a particularly high price. This is a primary motivation for doing as many operations per pass as possible. Our iburg-like rule matching works well for the pipeline of simple units found in standard OpenGL, but more complex units (as found in some new multitexture extensions, for example) require more powerful compiler technology. Two possibilities are surveyed by Harris [15].

5.2 Advantages

Our methodology allows research and development to proceed in parallel as shading languages, compilers, and hardware independently evolve. We can take advantage of the unique feature and performance needs of different application areas through specialized shading languages.

The application does not have to handle the complexities of multipass shading since the application interface is a scene graph. This model is a natural extension of most interactive applications, which already have a retained mode interface of some sort to enable users to manipulate their data. Applications still retain the other advantages of having a scene graph, like occlusion culling and level of detail management.

As mentioned, we have only implemented a few of the many possible compiler optimizations. As the compiler improves, our performance will improve, independent of language or hardware.

Finally, the rapid pace of graphics hardware development has resulted in systems with a diverse set of features and relative feature performance. Our design allows an application to use a shading language on all of the systems, and still take advantage of many of their unique characteristics. Hardware vendors do not need to create the shading compiler and retained data structures since they operate above the level of the drivers. Further, since complex effects can be supported on unextended hardware, designers are free to create fast, simple hardware without compromising on capabilities.

6 CONCLUSION

We have created a software layer between the application and the hardware abstraction layer to translate high-level shading descriptions into multi-pass OpenGL. We have demonstrated this approach with two examples, a constrained shading language that runs interactively on current hardware, and a fully general shading language. We have also shown that general shading languages, like the RenderMan Shading Language, can be implemented with only two additional OpenGL extensions.

There is a continuum of possible languages between ISL and the RenderMan Shading Language with different levels of functionality. We have applied our method to two different shading languages in part to demonstrate its generality.

There are many avenues of future research. New compiler technology can be developed or adapted for programmable shading. There are significant optimizations that we are investigating in our compilers. Research is also needed to understand what hardware features are best for supporting interactive programmable shading. Finally, given examples like the scientific visualization constructs described by Crawfis that are not found in the RenderMan shading language [9], we believe the wide availability of interactive programmable shading will spur exciting developments in new shading languages and new applications for them.

References

[1] BIRCH, P., BLYTHE, D., GRANTHAM, B., JONES, M., SCHAFER, M., SEGAL, M., AND TANNER, C. *An OpenGL++ Specification*. SGI, March 1997.

[2] BLYTHE, D., GRANTHAM, B., KILGARD, M. J., MCREYNOLDS, T., NELSON, S. R., FOWLER, C., HUI, S., AND WOMACK, P. Advanced graphics programming techniques using OpenGL: Course notes. In *Proceedings of SIGGRAPH '99* (July 1999).

[3] BOCK, D. Tech watch: Volume rendering. *Computer Graphics World 22*, 5 (May 1999).

[4] BRIGGS, P. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[5] CABRAL, B., CAM, N., AND FORAN, J. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *1994 Symposium on Volume Visualization* (October 1994), 91–98. ISBN 0-89791-741-3.

[6] CABRAL, B., OLANO, M., AND NEMEC, P. Reflection space image based rendering. *Proceedings of SIGGRAPH 99* (August 1999), 165–170.

[7] COOK, R. L. Shade trees. *Computer Graphics (Proceedings of SIGGRAPH 84) 18*, 3 (July 1984), 223–231. Held in Minneapolis, Minnesota.

[8] CORRIE, B., AND MACKERRAS, P. Data shaders. *Visualization '93 1993* (1993).

[9] CRAWFIS, R. A., AND ALLISON, M. J. A scientific visualization synthesizer. *Visualization '91* (1991), 262–267.

[10] DIEFENBACH, P. J., AND BADLER, N. I. Multi-pass pipeline rendering: Realism for dynamic environments. *1997 Symposium on Interactive 3D Graphics* (April 1997), 59–70.

[11] FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems 1*, 3 (September 1992), 213–226.

[12] GRITZ, L., AND HAHN, J. K. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools 1*, 3 (1996), 29–47.

[13] HAEBERLI, P. E., AND AKELEY, K. The accumulation buffer: Hardware support for high-quality rendering. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4 (August 1990), 309–318.

[14] HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4 (August 1990), 289–298.

[15] HARRIS, M. Extending microcode compaction for real architectures. In *Proceedings of the 20th annual workshop on Microprogramming* (1987), pp. 40–53.

[16] HART, J. C., CARR, N., KAMEYA, M., TIBBITTS, S. A., AND COLEMAN, T. J. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 1999), 45–53.

[17] HEIDRICH, W., AND SEIDEL, H.-P. Realistic, hardware-accelerated shading and lighting. *Proceedings of SIGGRAPH 99* (August 1999), 171–178.

[18] HEIDRICH, W., WESTERMANN, R., SEIDEL, H.-P., AND ERTL, T. Applications of pixel textures in visualization and realistic image synthesis. *1999 ACM Symposium on Interactive 3D Graphics* (April 1999), 127–134. ISBN 1-58113-082-1.

[19] JAQUAYS, P., AND HOOK, B. Quake 3: Arena shader manual, revision 10. In *Game Developer's Conference Hardcore Technical Seminar Notes* (December 1999), C. Hecker and J. Lander, Eds., Miller Freeman Game Group.

[20] KAUTZ, J., AND MCCOOL, M. D. Interactive rendering with arbitrary brdfs using separable approximations. *Eurographics Rendering Workshop 1999* (June 1999). Held in Granada, Spain.

[21] KELLER, A. Instant radiosity. *Proceedings of SIGGRAPH 97* (August 1997), 49–56.

[22] KYLANDER, K., AND KYLANDER, O. S. *Gimp: The Official Handbook*. The Coriolis Group, 1999.

[23] MAX, N., DEUSSEN, O., AND KEATING, B. Hierarchical image-based rendering using texture mapping hardware. *Rendering Techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)* (June 1999), 57–62.

[24] MCCOOL, M. D., AND HEIDRICH, W. Texture shaders. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 1999), 117–126.

[25] OLANO, M., HART, J. C., HEIDRICH, W., MCCOOL, M., MARK, B., AND PROUDFOOT, K. Approaches for procedural shading on graphics hardware: Course notes. In *Proceedings of SIGGRAPH 2000* (July 2000).

[26] OLANO, M., AND LASTRA, A. A shading language on graphics hardware: The PixelFlow shading system. *Proceedings of SIGGRAPH 98* (July 1998), 159–168.

[27] OPENGL ARB. Extension specification documents. http://www.opengl.org-/Documentation/Extensions.html, March 1999.

[28] PIXAR. *The RenderMan Interface Specification: Version 3.1*. Pixar Animation Studios, September 1999.

[29] SEGAL, M., AKELEY, K., FRAZIER, C., AND LEECH, J. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., 1999.

[30] SGI TECHNICAL PUBLICATIONS. *Cosmo 3D Programmer's Guide*. SGI Technical Publications, 1998.

[31] SIMS, K. Particle animation and rendering using data parallel computation. *Computer Graphics (Proceedings of SIGGRAPH 90) 24*, 4 (August 1990), 405–413.

[32] UPSTILL, S. *The RenderMan Companion*. Addison-Wesley, 1989.

Level-of-Detail Shaders

Marc Olano, Bob Kuehne *
SGI

Abstract

Current graphics hardware can render objects using simple procedural shaders in real-time. However, detailed, high-quality shaders will continue to stress the resources of hardware for some time to come. Shaders written for film production and software renderers may stretch to thousands of lines. The difficulty of rendering efficiently is compounded when there is not just one, but a scene full of shaded objects, surpassing the capability of any hardware to render. This problem has many similarities to the rendering of large models, a problem that has inspired extensive research in geometric level-of-detail and geometric simplification. We introduce an analogous process for shading, *shader simplification*. Starting from an initial detailed shader, shader simplification produces a new shader with extra level-of-detail parameters that control the shader execution. The resulting *level-of-detail shader*, can automatically adjust its rendered appearance based on measures of distance, size, or importance as well as physical limits such as rendering time budget or texture usage.

CR categories and subject descriptors: I.3.3 [Computer Graphics]: Picture/Image generation — Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism — Color, shading, shadowing and texture.

Keywords: Interactive Rendering, Rendering Systems, Hardware Systems, Procedural Shading, Languages, Multi-Pass Rendering, Level-of-Detail, Simplification, Computer Games, Reflectance & Shading Models.

1 INTRODUCTION

Procedural shading is a powerful technique, first explored for software rendering in work by Cook and Perlin [10, 35], and popularized by the RenderMan Shading language [20]. A shader is a simple procedure written in a special purpose high-level language that controls some aspect of the appearance of an object to which it is applied. The term *shader* is used generically to refer to procedures that compute surface color, attenuation of light through a volume (as with fog), light color and direction, fine changes to the surface position, or transformation of control points or vertices.

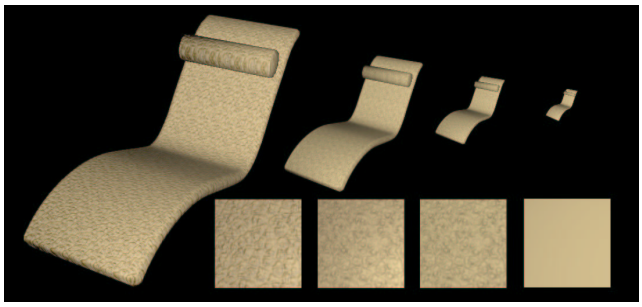


Figure 1: LOD shader upholstering a Le Corbusier chair.

Recent graphics hardware can render simple procedural shaders in real-time [4, 5, 31, 33, 34, 36]. Shaders that exceed the hardware's abilities for rendering of a single object must be rendered using multiple passes through the graphics pipeline. The resulting multi-pass shaders can achieve real-time performance, but many complex shaders in a single scene can easily overwhelm any graphics hardware. Even for shaders that execute in a single rendering pass, the number of textures or combiner stages used can affect overall performance [31].

Consider a realistic shader for a leather chair. Features of this shader may include an overall leather texture or bump map, a couple of measured BRDFs (bidirectional reflectance distribution functions) for worn and unworn areas on the seat, bumps for the stitching, with dust collected in the crevices, scuff marks, changes in color due to variations in the leather, and potentially even more. Such a shader can provide a satisfying interactive rendering of the seat for detailed examination, but is overkill as you move away to see the rest of the room and all the other, buildings, trees and pedestrians using shaders of similar complexity. Figure 1 does not have all the features described, but with a bump map and measured leather BRDF it still exceeds current single pass rendering capabilities.

In this paper, we introduce level-of-detail shaders (LOD shaders) to solve the problem of providing both interactive performance and convincing detailed shading of many objects in a scene. A level-of-detail shader automatically adjusts the shading complexity based on one or more input pa-

*email:{olano,rpk}@sgi.com

rameters, providing only the detail appropriate for the current viewing conditions and resource limits. We present a general framework for creating a level-of-detail shader from a detailed source shader which could be used for automatic LOD shader generation. Finally, we provide details and results from our building-block based level-of-detail shader tools, where the general framework for shader simplification has been manually applied to building-block functions used for writing complex shaders.

1.1 Background

This work is directly inspired by the body of research on geometric simplification. Specifically, many of our shader simplification operations are modeled after operations from the topology-preserving geometric level-of-detail literature. Schroeder and Turk both performed early work in automatic mesh simplification using a series of local operations, each resulting in a smaller total polygon count for the entire model [39, 41]. Hoppe used the collapse of an edge to a single vertex as the basic local simplification operation. He also introduced progressive meshes, where all simplified versions of a model are stored in a form that can reconstructed to any level at run-time [24]. These ideas have had a large influence on more recent polygonal simplification work ([16, 22, 25] and many others).

Many shader simplifications involve generating textures to stand in for one or more other shading operations. Guenter, Knoblock and Ruf replaced static sequences of shading operations with pre-generated textures [19]. Heidrich has analyzed texture sizes and sampling rates necessary for accurate evaluation of shaders into texture [32]. In a related vein, texture-impostor based simplification techniques replace geometry with pre-rendered textures, either for indoor scenes as has been done by Aliaga [2] or outdoor scenes as by Shade et al. [40].

We also draw on the body of BRDF approximation methods. Like shading functions, BRDFs are positive everywhere. Fournier used singular value decomposition (SVD) to fit a BRDF to sums of products of functions of light direction and view direction for use in radiosity [13]. Kautz and McCool presented a similar method for real-time BRDF rendering, computing functions of view, light, or other bases as textures using either SVD or a simpler normalized integration method [27]. McCool, Ang and Ahmad's homomorphic factorization uses only products of 2D texture lookups, fit using least-squares [29]. In a related area, Ramamoorthi and Hanrahan used a common set of spherical harmonic basis textures for reconstructing irradiance environment maps [37].

This work is also directly derived from efforts to antialias shaders. The primary form of antialiasing provided in the RenderMan shading language is a manual transformation of the shader, relying on the shader-writer's knowledge to effectively remove high-frequency components of the shader or smooth the sharp transitions from an `if`, by instead using a `smoothstep` (cubic spline interpolation between two values) or `filterstep` (smoothstep across the current sample width) [11]. Perlin describes automatic use of blending where `if` is used in the shading code [11]. Heidrich and his collaborators also did automatic antialiasing, using affine arithmetic to compute the shading results and estimate the frequency and error in the results [23].

Finally, there have been several researchers who have done more ambitious shader transformations. Goldman described multiple versions of a fur shader used in several movies, though switches between *realfur* and *fakefur* were only done between shots [18]. Kajiya was the first to pose the problem of converting large-scale surface characteristics to a bump map or BRDF representation [26]. Along this line, Fournier used nonlinear optimization to fit a bump map to a sum of several standard Phong *peaks* [12]. Cabral, Max and Springmeyer addressed the conversion from bump map to BRDF through a numerical integration pre-process [7], and Becker and Max solved it for conversion from RenderMan-based displacement maps to bump maps and then to a BRDF representation [6]. More recently, Apodaca and Gritz manually created a hierarchy of filtered level-of-detail textures [3], while Kautz approached the problem in reverse, creating bump maps to statistically match a chosen fractal micro-facet BRDF [28].

This work is set within the context of recent advances in interactive shading languages, motivating the need for shaders that can transition smoothly from high quality to fast rendering. The first such system by Rhoades et al. was a relatively low-level language for the Pixel-Planes 5 machine at UNC [38]. This was followed by Olano and collaborators with a full interactive shading language on UNC's PixelFlow system [33]. Peercy and coworkers at SGI created a shading language that runs using multiple OpenGL Rendering passes [34]. The work presented here uses their OpenGL Shader ISL language as the format for both input shaders and LOD shader results.

There are many emerging options for assembler-level interfaces to hardware accelerated shading, including offerings by NVIDIA and ATI as well as a shading interface within DirectX [4, 5, 30, 31]. The shading group at Stanford, led by Kekoa Proudfoot and Bill Mark, created another high-level real-time shading language that can be compiled into either multiple rendering passes or a single pass using NVIDIA or ATI hardware extensions [36]. A group at 3DLabs, led by Randi Rost, is also spearheading an effort to create a high-level shading language for OpenGL version 2.0.

2 USING LOD SHADERS

Using a single LOD shader that encapsulates the progression of levels of detail provides many of the advantages for simplified shaders that progressive meshes provide for geometry. The following directly echos the points from Hoppe's original progressive mesh paper [24].

- *Shader simplification*: The LOD shader can be generated automatically from an initial complex shader using automatic tools (though as in the early days of mesh

simplification, these tools are not yet as automatic as we would like).

- *LOD approximation*: Like a progressive mesh, an LOD shader contains all levels of detail. Thus it can include the shader equivalent of Hoppe's *geomorphs* to smoothly transition from one level to the next.

- *Progressive transmission and compression*: The representation of a shader is much smaller than that of a mesh. Even relatively complex RenderMan shaders are typically only a few thousand lines of code. Shaders for real-time are seldom more complex than several tens of lines of code. Yet a scene with thousands of LOD shaders may still benefit by first storing and sending the simplest levels followed by transmission of the more complex levels.

- *Selective Refinement*: Selective refinement for meshes refers to simplifying some portions of the mesh more than others based on current viewing conditions, encompassing both variation across the object and a guided decision on which of the stored simplifications to apply. For an LOD shader these aspects are treated independently. Current hardware does not realize any benefit from shading variations across a single object, but a single LOD shader will present a high quality appearance on some surfaces while using a lower quality for others, based on distance, viewing angle or other factors. The LOD shader may also apply certain simplifications and not others based on pressure from hardware resource limits. For example, if available texture memory is low, texture-reducing simplification steps may be applied in one part of the shader while leaving more computation-heavy portions of the shader to be rendered at full detail.

Many of these points depend on the storage of an LOD shader. Starting from a complex shader we create a series of simplification operations to produce the most simplified shader, represented as another shader in the source shading language. This combined shader includes all of the levels within a single shading function with additional level control parameters. This provides several practical advantages as the LOD shader is indistinguishable, beyond its additional parameters, from a non-LOD shader. Since OpenGL Shader (and most other shading systems) set shader parameters by name, with default values for unset parameters, LOD shaders are easily interchanged with other shaders. For example, this can allow easy drop in replacement of the covering on a car seat, from a simple stand-in to a non-LOD vinyl shader, an LOD leather shader, or an LOD fabric shader.

The set of level-control parameters are the one aspect that distinguishes the interface to an LOD shader from other shaders. For interchangeable use the parameter set should be agreed upon by both the application and shader simplifier. These parameters are used within the LOD shader to switch

```
FB=diffuse();

FB*=texture("tex");
```

a) basic block

```
FB=diffuse();
if (time<10)
  FB*=texture("tex");
```

b) split blocks

Figure 2: Candidate blocks. a) a single basic block that could be simplified. b) blocks split by a conditional — will not be merged together

and blend between different levels as well as to define the ranges where each level is valid. As with geometric level-of-detail, parameter choices may include distance to the object, approximate screen size of the rendered object, importance of the object, or available time budget. For shading, we may also add budgets for hardware resource limits such as texture memory availability. Many of these parameters could instead be collected into a single aggregate parameter, or controlled through an optimization function as done by Funkhouser and Séquin [15]. All examples in this paper use a single parameter set using a distance metric.

3 SIMPLIFICATION FRAMEWORK

Shader simplification creates an LOD shader from an arbitrary source shader. We describe the simplification process in terms of four stages. First, identify candidate blocks of shader code. Second, produce a set of simplified versions of the candidate blocks. Third, associate level parameters with the simplified blocks, and finally assemble the result into an LOD shader. These stages can be repeated to achieve further simplification, where two or more simplified blocks can be combined into a single larger candidate block for another simplification run.

3.1 Finding Candidate Blocks

The first step toward creating an LOD shader is identifying blocks of shader code that are candidates for simplification. These are like edges for edge-collapse based polygonal simplification. Finding the set of candidate blocks in a shader is slightly more complicated than finding the set of edges in a model, but can be done with a static analysis of the original shader code.

A static analysis is one done before actual execution; it only has access to what can be inferred from the source code itself. In particular, results for conditionals and loops involving compile-time constants are known (*uniform* in ISL parlance), but not ones that might change at run-time (*parameter* in ISL). As a result, choosing a static analysis restricts simplification possibilities to what can be done within a basic block, without crossing a run-time loop or conditional (Figure 2).

Each block within the shader has some variables that are input to the computations within the block and others that are results computed by the block. Expressions within the block form a dependence graph with operations represented as nodes in the graph and variables as edges linking operation to operation. This graph can be partitioned into subgraphs

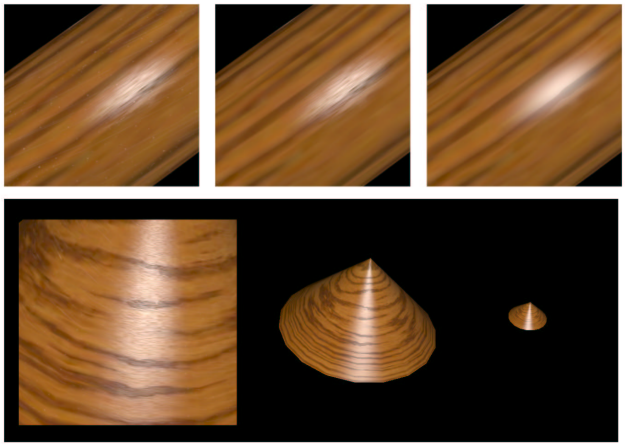


Figure 3: Removal of operations as contributions become imperceptible. Top row, left to right: Close-up of torus mapped with detail dust and scratch textures, with dust and scratches removed, with specular mask removed. Bottom row, left to right: image sequence of the wood applied to a cone with each removal displayed at it's expected switching distance.

where each subgraph computes one block output or intermediate result. These subgraphs are the candidate blocks for simplification. Any basic block can be partitioned in many ways, and the choice of block partitioning is somewhat analogous to choosing edges for mesh simplification.

3.2 Simplifications

Each of the candidate blocks described above computes one result based on a set of inputs. The simplification operations on this block perform a local substitution of a simpler form in place of the original, producing equivalent output while keeping the form of the total shader the same. Simplifications that are not lossy are handled by the shading compiler optimization [19, 33, 34, 36].

Simplifications are chosen by matching a set of heuristic rules. While logically separate, the selection of simplification rules and partitioning of the basic block can be done at the same time using a tool like `iburg` [14]. `Iburg` is a compiler tools designed for use in code generation. Given a piece of code represented as an expression tree, it finds the least cost cover by a set of rules through a bottom-up dynamic programming algorithm.

Finding simplification rule costs for use by `iburg` requires analysis of input textures as well as the shader itself, and application of a rule may require generating a new derived texture as part of the LOD shader generation pre-process.

We classify these rule-based substitutions into one of four forms.

Remove: A candidate block that doesn't contribute enough anymore, or that consists of only high-frequency elements above the Nyquist frequency is replaced by a constant. This effectively removes the effect of portions of the shader that are no longer significant (Figures 3,4).

Collapse: A candidate block consisting of several operations may be merged into a single new operation. For example, a coarse texture and a rotated and repeated detail texture can be combined into a single merged texture of a new size (Figure 5).

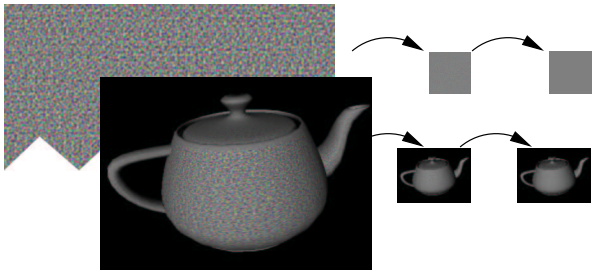


Figure 4: Band-limited Perlin noise texture, noise at a distance, and noise replaced with average value

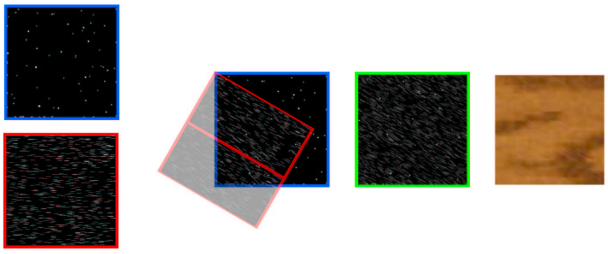


Figure 5: Collapsing two texture operations into a single texture. Left to right, the two initial textures, the two textures transformed and overlaid, the collapsed texture result, and an example of the collapsed texture in use as dust and scratch wood detail.

Substitute: A candidate block identified as implementing a known shading method may be replaced by a simpler method with similar appearance. For example, a bump map can be replaced by a gloss map to modulate the highlight intensity, or a simple texture map (Figure 6). A texture indexed by the surface normal is probably part of a lighting model, and depending on the contents of the texture, may be replaced by the built-in diffuse lighting model. Similarly, a texture indexed by the half angle vector (norm$(V+L)$ for view vector V and light vector L) is a candidate for replacement by one or more applications of the built-in Phong specular model. A texture can be replaced by a smaller low-pass filtered version of the texture and a constant representing the removed high-frequency terms.

Approximate: Approximation rules treat the candidate block as a general function to be approximated. They can theoretically be applied to any block, though not always as effectively as the application-specific rules.

While a variety of function approximation methods are possible, we have focused on ones developed for BRDF approximation [27, 29]. As these methods are texture-based, they are most useful when total texture usage is not the limiting factor. Two issues prevent our approximation rules from being more generally useful, though we believe they are aspects of the approximations we chose to explore and not all applicable function approximation methods.

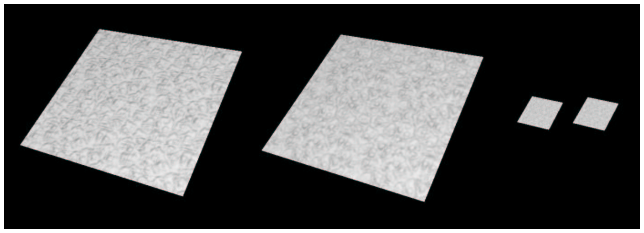


Figure 6: Replacing a bump map with a texture. Left to right, the original bump map, the bump texture at full scale, and the bump map and texture at the expected switching distance.

First, these approximations are based on a factorization into products or sums of products of functions of two variables that can be stored in a texture. In the right coordinate system, BRDFs are well suited to this factorization, usually requiring only one or two terms. Automatic simplification calls for automatic determination of a coordinate system. Arbitrary shading expressions can also be poorly suited to such a factorization in any coordinate system, allowing no acceptable approximation by the homomorphic factorization method, or needing so many SVD terms as to become more expensive than the original expression.

Second, the least squares or singular value decomposition problems are stated in terms of matrices with a number of rows and columns equal to the total number of texels in each approximating texture. Computing these textures rapidly scales to gigabytes, even for modest component texture sizes. Worse, we want to speculatively compute the approximations to evaluate their fitness. The original application to BRDFs limited the component texture sizes to 32x32 or 64x64 resulting in computations with 1024x1024 to 4096x4096 matrices.

3.3 Level Parameters

Selection of simplified verses unsimplified blocks is based on one or several level parameters. For example, switching from a band-limited noise texture to a constant value should happen when the changes in the noise texture are no longer visible (Figure 4). That point can be approximated based either on the distance or screen size of the object. The same transition can also be triggered by a lack of available rendering time, or a lack of available texture memory to store the noise texture.

To manage these different level parameters, we can associate a range for each parameter with each simplified block. Using the noise example above, a constant should be used instead of the noise texture whenever the available texture memory is less than the size of the texture, or there is not enough time to render another texture, or the expected mapping to screen pixels will blur the band-limited noise away.

3.4 Assemble

Given the simplified blocks and level parameter ranges, it is straightforward to assemble them with appropriate conditionals into an LOD shader. Rendering-metric level parameters, like distance or screen coverage, are shared by all blocks in the shader, each emitting a statement of the form

```
if(distance < low_threshold)
  do_simplified_block
else if(distance < high_threshold)
  do_transition_block
else
  do_original_block
```

For resource-accounting level parameters (e.g. available time or texture memory) the blocks are prioritized, and comparisons are emitted for the total consumed by this block and all higher priority blocks.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	36.4, 27.6	44.5, 34.4	—, —
Wood (Remove)	18.4, 11.6	18.9, 11.9	19.1, 64.3
Leather (Replace)	25.4, 14.1	43.7, 25.3	79.8, 64.3

Table 1: Result times for test LOD shaders on the 1772 triangle chair model performed on an SGI Octane MXE. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	52.9, 33.8	68.2, 42.1	—, —
Wood (Remove)	20.7, 9.2	23.0, 10.0	25.2, 10.7
Leather (Replace)	30.7, 12.3	55.2, 22.8	140.9, 80.3

Table 2: Result times for test LOD shaders on a 3280 triangle draped cloth model consisting of 40 length-82 triangle strips, performed on an SGI Octane MXE. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

4 RESULTS

We have described a general theory of shader simplification. Our current results are a modest start within this framework. Specifically, we have produced a set of LOD-aware building block functions for shader construction. This style of shader writing is similar to Abram and Whitted's graphical building-block shader system [1]. Example building-blocks include bump map, a BRDF model, Fresnel reflectance, or noise or turbulence textures with a lookup as used by Hart [21].

Our LOD blocks were created by manually following the steps described in our simplification framework: identify candidate blocks within a building block function, apply one of the simplification rules described in Section 3.2, associate it with a range of an aggregate level parameter, and create conditional blocks for the original code, transition code and simplified code. Despite the manual simplification, we call this semi-automatic because any shaders written using the building blocks, either knowing about level-of-detail or not, become LOD shaders by switching to the LOD building blocks.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	9.2, 11.2	11.8, 14.0	—, —
Wood (Remove)	3.6, 5.3	4.1, 5.8	4.5, 6.5
Leather (Replace)	6.4, 8.8	14.7, 18.7	27.7, 35.7

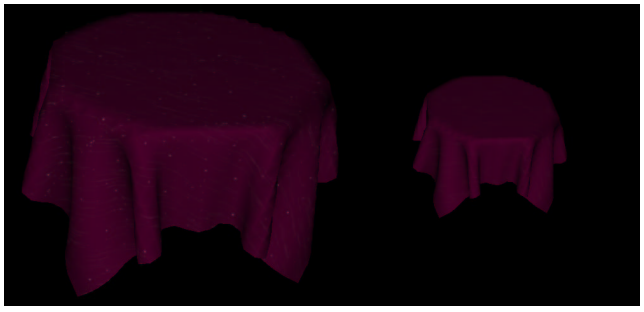
Table 3: Result times for test LOD shaders on the 1772 triangle chair model performed on an SGI O2. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	13.6, 15.9	18.2, 20.4	—, —
Wood (Remove)	4.9, 6.9	5.4, 7.6	6.0, 8.5
Leather (Replace)	8.1, 10.3	19.8, 23.9	40.3, 52.3

Table 4: Result times for test LOD shaders on the 3280 triangle draped cloth model performed on an SGI O2. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

Tables 1–4 show LOD shader timing in frames per second for several sample LOD shaders. Each shader demonstrates several transitions of specific LOD simplification operations. The Wood shader used in these tests first removes an overlay scratch texture, then removes a specular masking operation, creating three levels-of-detail. Figure 3 shows the removal LOD sequence. The Plastic shader demonstrates the collapse simplification by taking two textures, each applied with its own transformation, and merging these two separate texture passes in a third texture. This resultant texture is then used to shade the object in a single texture for lower levels-of-detail as shown in Figures 5 and 7. The Leather shader demonstrates the replace simplification in the first level-of-detail by replacing a true bump map with a simple texture. The second level in the Leather removes the texture with a simple constant color. Results of this operation sequence are seen in Figure 9.

An overview of the performance results shows much what we would expect — that less detailed shaders result in faster overall rendering. However, as the different results indicate, the shading operations are not purely fill-limited, and rendering nearly 4x fewer pixels in certain cases results in only a modest performance improvement. As certain passes occur, the object's geometry is also re-rendered, yielding a coupling between type of rendering passes constructed for a particular



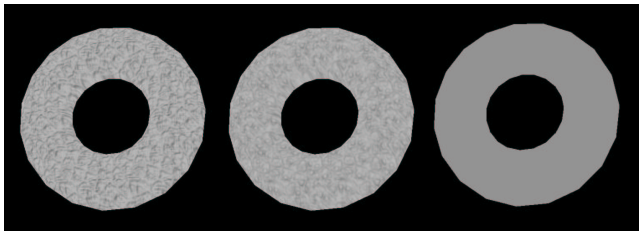


Figure 8: Two replace simplifications in a bumpy leather shader.

shader and that shader's. This implies that LOD shaders can accomplish only part of the task, and should also be accompanied by geometric simplification.

5 CONCLUSIONS AND FUTURE WORK

We have presented LOD shaders: procedural shaders that automatically adjust their level of shading detail for interactive rendering. We also presented a general framework for shader simplification — the process of creating LOD shaders from an ordinary shader. This framework is sufficiently general to serve as a guide for manual shader simplification or as a basis for automatic simplification. Finally, we presented our results for semi-automatic shader simplification using manually generated shading function building blocks for SGI's OpenGL Shader. These LOD shader building blocks implement the same functions as building blocks already provided with OpenGL Shader, but with added level-of-detail parameters to control aspects of their shading complexity.

In the future, we would like to create tools for fully automatic shader simplification. Our current simplification framework also only considers a static analysis of the shader for simplification. Following the lead of texture-based simplification researchers like Aliaga and Shade et al., we could generate new textures on the fly warping them for use over several frames or updating when they become too different [2, 40].

Logically, it should be possible to generalize our remove, collapse and substitution rules into a more widely applicable approximation rule form. Other function fitting methods should be tried to make the approximation rules more useful.

Since rendering with LOD shaders will usually be accompanied by geometric level-of-detail, they should be more closely linked. Cohen et al. Garland and Heckbert and others have shown that geometric simplification can be affected by appearance [8, 17]. Shader simplification should also be affected by geometric level-of-detail (e.g. whether per-vertex Phong shading is a good substitute for a texture-based illumination depends on how the object is tessellated).

Finally, we provide no guarantees on the fidelity of our simplifications. Many geometric simplification algorithms have been successful without providing exact error metrics or bounds. However, algorithms such as simplification envelopes by Cohen et al. provide hard bounds on the amount of error introduced by a simplification [9], guarantees that are important for some users. Further investigation is neces-

6 ACKNOWLEDGMENTS

The Le Corbusier chair was modeled by Jad Atallah, JLA Studio and distributed by 3dcafe.com. The Porsche data was distributed by 3dcafe.com. The leather BRDF is from Michael McCool, fit by homomorphic factorization to data from the Columbia-Utrecht Reflectance and Texture Database. The car paint BRDF also from Michael McCool, fit to data for Dupont Cayman lacquer from the Ford Motor Company and measured at Cornell University.

We'd also like to thank Dave Shreiner for his helpful comments on the drafts paper.

References

[1] ABRAM, G. D., AND WHITTED, T. Building block shaders. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (Dallas, Texas, August 1990), vol. 24, pp. 283–288. ISBN 0-201-50933-4.

[2] ALIAGA, D. G. Visualization of complex models using dynamic texture-based simplification. In *IEEE Visualization '96* (October 1996), IEEE, pp. 101–106. ISBN 0-89791-864-9.

[3] APODACA, A. A., AND GRITZ, L. *Advanced RenderMan*, first ed. Morgan Kaufmann, 2000.

[4] ATI. *Pixel Shader Extension*, 2000. Specification document, available from http://www.ati.com/online/sdk.

[5] ATI. *Vertex Shader Extension*, 2001. Specification document, available from http://www.ati.com/online/sdk.

[6] BECKER, B. G., AND MAX, N. L. Smooth transitions between bump rendering algorithms. In *Proceedings of SIGGRAPH 93* (Anaheim, California, August 1993), Computer Graphics Proceedings, Annual Conference Series, pp. 183–190. ISBN 0-201-58889-7.

[7] CABRAL, B., MAX, N., AND SPRINGMEYER, R. Bidirectional reflection functions from surface bump maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)* (Anaheim, California, July 1987), vol. 21, pp. 273–281.

[8] COHEN, J., OLANO, M., AND MANOCHA, D. Appearance-preserving simplification. In *Proceedings of SIGGRAPH 98* (Orlando, Florida, July 1998), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 115–122. ISBN 0-89791-999-8.

[9] COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., JR., F. P. B., AND WRIGHT, W. Simplification envelopes. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 119–128. ISBN 0-201-94800-1.

[10] COOK, R. L. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (Minneapolis, Minnesota, July 1984), vol. 18, pp. 223–231.

[11] EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. *Texturing and Modeling*, second ed. Academic Press, 1998.

[12] FOURNIER, A. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination* (May 1992), Canadian Information Processing Society, pp. 45–52.

[13] FOURNIER, A. Separating reflection functions for linear radiosity. In *Proceedings of Eurographics Workshop on Rendering* (Dublin, Ireland, June 1995), pp. 296–305.

[14] FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems 1*, 3 (September 1992), 213–226.

[15] FUNKHOUSER, T. A., AND SÉQUIN, C. H. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93* (Anaheim, California, August 1993), Computer Graphics Proceedings, Annual Conference Series, pp. 247–254. ISBN 0-201-58889-7.

[16] GARLAND, M., AND HECKBERT, P. S. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 209–216. ISBN 0-89791-896-7.

[17] GARLAND, M., AND HECKBERT, P. S. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization '98* (October 1998), IEEE, pp. 263–270. ISBN 0-8186-9176-X.

[18] GOLDMAN, D. B. Fake fur rendering. In *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 127–134. ISBN 0-89791-896-7.

[19] GUENTER, B., KNOBLOCK, T. B., AND RUF, E. Specializing shaders. In *Proceedings of SIGGRAPH 95* (Los Angeles, California, August 1995), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 343–350. ISBN 0-201-84776-0.

[20] HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (Dallas, Texas, August 1990), vol. 24, pp. 289–298. ISBN 0-201-50933-4.

[21] HART, J. C., CARR, N., KAMEYA, M., TIBBITTS, S. A., AND COLEMAN, T. J. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (Los Angeles, California, August 1999), ACM SIGGRAPH / Eurographics / ACM Press, pp. 45–53.

[22] HECKBERT, P., ROSSIGNAC, J., HOPPE, H., SCHROEDER, W., SOUCY, M., AND VARSHNEY, A. Multiresolution surface modeling. In *SIGGRAPH 1997 Course Notes* (August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley.

[23] HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. Sampling procedural shaders using affine arithmetic. 158–176. ISSN 0730-0301.

[24] HOPPE, H. Progressive meshes. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 99–108. ISBN 0-201-94800-1.

[25] HOPPE, H. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 189–198. ISBN 0-89791-896-7.

[26] KAJIYA, J. T. Anisotropic reflection models. In *Computer Graphics (Proceedings of SIGGRAPH 85)* (San Francisco, California, July 1985), vol. 19, pp. 15–21.

[27] KAUTZ, J., AND MCCOOL, M. D. Interactive rendering with arbitrary brdfs using separable approximations. In *Eurographics Rendering Workshop 1999* (Granada, Spain, June 1999), Springer Wein / Eurographics.

[28] KAUTZ, J., AND SEIDEL, H.-P. Towards interactive bump mapping with anisotropic shift-variant brdfs. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 2000), 51–58.

[29] MCCOOL, M. D., ANG, J., AND AHMAD, A. Homomorphic factorization of brdfs for high-performance rendering. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 171–178. ISBN 1-58113-292-1.

[30] MICROSOFT. *DirectX Graphics Programmers Guide*, directx 8.1 ed. Microsoft Developers Network Library, 2001.

[31] NVIDIA. *NVIDIA OpenGL Extensions Specifications*, March 2001.

[32] OLANO, M., HART, J. C., HEIDRICH, W., LINDHOLM, E., MCCOOL, M., MARK, B., AND PERLIN, K. Real-time shading. In *SIGGRAPH 2001 Course Notes* (August 2001).

[33] OLANO, M., AND LASTRA, A. A shading language on graphics hardware: The pixelflow shading system. In *Proceedings of SIGGRAPH 98* (Orlando, Florida, July 1998), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 159–168. ISBN 0-89791-999-8.

[34] PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000* (July 2000), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 425–432. ISBN 1-58113-208-5.

[35] PERLIN, K. An image synthesizer. In *Computer Graphics (Proceedings of SIGGRAPH 85)* (San Francisco, California, July 1985), vol. 19, pp. 287–296.

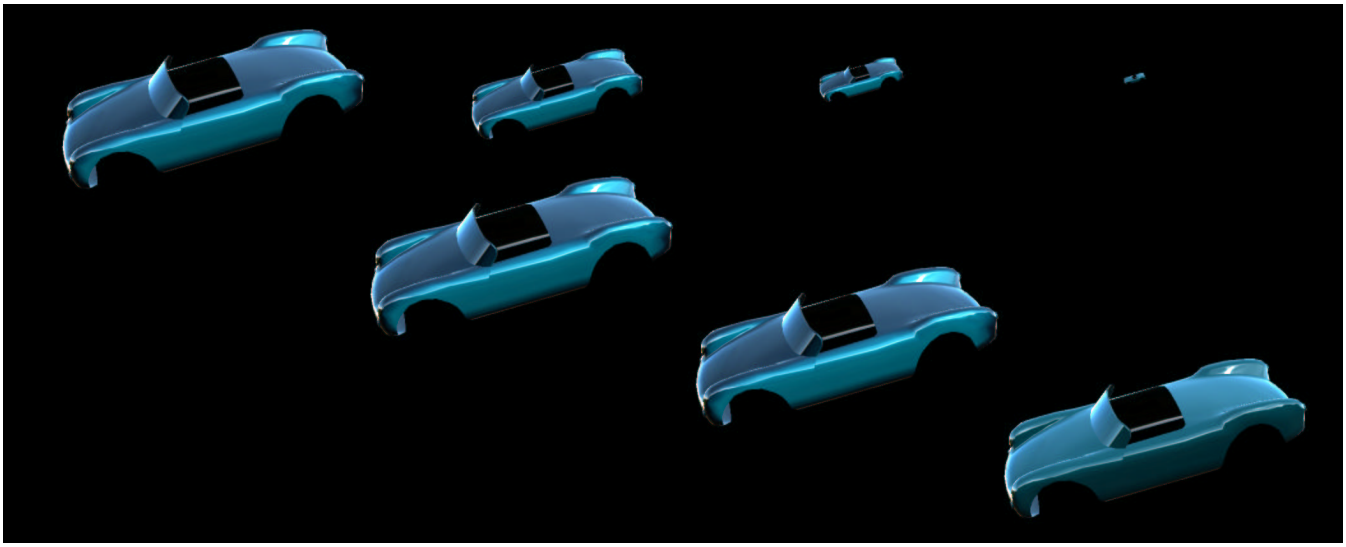


Figure 9: Car paint LOD shader using LOD versions of OpenGL Shader's `microfacetBRDF` and `hdrFresnel` building block functions.

[36] PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 159–170. ISBN 1-58113-292-1.

[37] RAMAMOORTHI, R., AND HANRAHAN, P. An efficient representation for irradiance environment maps. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 497–500. ISBN 1-58113-292-1.

[38] RHOADES, J., TURK, G., BELL, A., STATE, A., NEUMANN, U., AND VARSHNEY, A. Real-time procedural textures. In *1992 Symposium on Interactive 3D Graphics* (March 1992), vol. 25, pp. 95–100. ISBN 0-89791-467-8.

[39] SCHROEDER, W. J., ZARGE, J. A., AND LORENSEN, W. E. Decimation of triangle meshes. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (Chicago, Illinois, July 1992), vol. 26, pp. 65–70. ISBN 0-201-51585-7.

[40] SHADE, J., LISCHINSKI, D., SALESIN, D. H., DEROSE, T. D., AND SNYDER, J. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 75–82. ISBN 0-201-94800-1.

[41] TURK, G. Re-tiling polygonal surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (Chicago, Illinois, July 1992), vol. 26, pp. 55–64. ISBN 0-201-51585-7.

Interactive Shading Language (ISL) Language Description Version 2.4 March 26, 2002

Contents

I. Introduction

ISL is a shading language designed for interactive display. Like other shading languages, programs written in ISL describe how to find the final color for each pixel on a surface. ISL was created as a simple restricted shading language to help us explore the implications of interactive shading. As such, the language definition itself changes often. While this may be a snapshot specification for ISL, ISL is **not** proposed as a formal or informal language standard. Shading language design for interactive shading is still an open area of research.

A. Features in common with other shading languages

The final pixel color comes from the combined effects of two function types. A *light*

shader computes the color and intensity for a light hitting the surface. Light shaders can be used for ambient, distant and local lights. Several light shaders may be involved in finding the final color for a single pixel. A *surface shader* computes the base surface color and the interaction of the lights with that surface. The term *shader* is used to refer to either of these special types of function.

All shading code is written with a single instruction, multiple data (SIMD) model. ISL shaders are written as if they were operating on a single point on the surface, in isolation. The same operations are performed for all pixels on the surface, but the computed values can be different at every pixel.

Like other shading languages that follow the SIMD model, ISL data may be declared *varying* or *uniform*. Varying values may vary from pixel to pixel, while uniform values must be the same at every pixel on the surface.

B. Major differences from other shading languages

ISL has several differences and limitations that distinguish it from more full-featured shading languages:

- The primary varying data type in ISL is limited to the range [0,1]. Results outside this range are clamped.
- ISL does not allow texture lookups based on computed results.
- ISL does not allow user-defined parameters that vary across the surface. Such parameters must either be computed or loaded as texture.

ISL is also different from most other shading languages in that more than one surface shader may be applied to each surface. The shaders are applied in turn and may composite or blend their results. ISL no longer supports explicit atmosphere shaders. Any light transmission effects between the surface and eye can be handled in the final shader applied to each surface.

II. Files

The appearance of a shaded surface is defined by one or more ISL surface shaders and possibly one or more ISL light shaders. Each shader is defined in its own ISL source files, which should have the file name extension .isl.

A. File contents

Only one shader definition (whether light or surface) can appear in each .isl file. The .isl file may include C preprocessor-like `#include` directives to get access to functions or global variable definitions stored in another file.

Comments in isl may be either C or C++-style (`/*comment*/` or `// comment to end of line`)

B. File compilation

There are two ways to compile a set of ISL files into the rendering passes used to compute surface appearance. The first is to use the ISL run-time library. The second is to use the command line compiler and translator. Both are documented in the shader(1) man page. The ISL Library consists of a set of C++ classes that enable an application to compile that appearance consisting of ISL shaders into an OpenGL stream. The compiled appearance can be associated with geometry from the application, and rendered to an OpenGL rendering context opened by the application. The ISL compiler, islc, converts a set of ISL files into a pass description (.ipf) file. Information on running islc can be found on the islc(1) man page. The pass description file can be converted either to C OpenGL code with the command line translator ipf2ogl (see the ipf2ogl(1) man page), or to a Performer pass file with the command line translator ipf2pf (shipped with Performer 2.4 or later).

III. Data types

All ISL data is classified as either *varying*, *parameter* or *uniform*. Varying data may hold a different value at each pixel. Parameter data must have the same value at every pixel on a surface, but can differ from surface to surface or from frame to frame. Changes to varying or parameter data do not require recompiling the shader. Uniform data also has the same value at every pixel on the surface, but changes to uniform data only take effect when the shader is recompiled.

The complete list of ISL data types is:

uniform float *uf* **parameter float** *pf*	*uf* and *pf* are each a single floating point value
uniform color *uc* **parameter color** *pc*	*uc* and *pc* are each a set of four floating point values, representing a color, vector or point. For colors, the components are ordered red, green, blue and alpha. For points, the components are ordered x,y,z and w.
varying color *vc*	*vc* is a four element color, vector or point that may have different values at each pixel on the surface. Elements of the color are constrained to lie between 0 and 1. Negative values are clamped to zero and values greater than one are clamped to one
uniform matrix *um* **parameter matrix** *pm*	*um* and *pm* are each a set of sixteen floating point values, representing a 4x4 matrix in row-major order (all four elements of first row, all four elements of second row, ...)
uniform string *us*	*us* is a character string, used for texture names.

ISL also allows 1D arrays of all uniform and parameter types, using a C-style specification:

uniform float *ufa*[*n*]	*ufa* is an array with *n* uniform float point elements, *ufa[0]* through *ufa[n-1]*
parameter float *pfa*[*n*]	*ufa* is an array with *n* parameter float point elements, *pfa[0]* through *pfa[n-1]*
uniform color *uca*[*n*]	*uca* is an array with *n* uniform color elements, *uca[0]* through *uca[n-1]*.
parameter color *uca*[*n*]	*pca* is an array with *n* parameter color elements, *pca[0]* through *pca[n-1]*.
uniform matrix *uma*[*n*]	*uma* is an array with *n* uniform matrix elements, *uma[0]* through *uma[n-1]*
parameter matrix *pma*[*n*]	*pma* is an array with *n* parameter matrix elements, *pma[0]* through *pma[n-1]*
uniform string *usa*[*n*]	*usa* is an array with *n* uniform string elements, *usa[0]* through *usa[n-1]*

IV. Variables and identifiers

Identifiers in ISL are used for variable or function names. They begin with a letter, and may be followed by additional letters, underscores or digits. For example a, abc, C93d, and d_e_f are all legal identifiers.

Several variables are predefined with special meaning:

varying color FB	Current frame buffer contents. This is the intermediate result location for almost all varying operations.
parameter matrix shadermatrix	Arbitrary matrix associated with the shader at compile time. This may be used to control the coordinate space where the shader operates.
parameter color lightVector	Within a light shader, the direction the light is shining. This vector may be modified by the light shader. Within a surface shader, the direction of the most recent light.
uniform float pi	The math constant.
uniform float numambientlights	Number of ambient lights in the current islAppearance.
uniform float numdirectlights	Number of direct lights (= both local and distant lights) in the current islAppearance.

V. Uniform operations

In the following, *uf* and *uf0*- *uf15* are uniform floats; *ufa* is an array of uniform floats; *uc*, *uc0* and *uc1* are uniform colors; *uca* is an array of uniform colors; *um*, *um0* and *um1* are uniform matrices; *uma* is an array of uniform matrices; *us*, *us0* and *us1* are uniform strings; *usa* is an array of uniform strings; and *ur*, *ur0* and *ur1* are uniform relations.

A. uniform float

Operations producing a uniform float:

variable reference	Value of uniform float variable.
float constant	One of the following non-case-sensitive patterns: **0x*H*** (hex integer); **0*O*** (octal integer); ***D***; ***D.***; ***.D***; ***D.D***; ***D*e*S*D***; ***D.*e*S*D***; ***.D*e*S*D***; ***D.D*e*S*D*** Where ***H*** = 1 or more hex digits (0-9 or a-f) ***O*** = 1 or more octal digits (0-7) ***D*** = 1 or more decimal digits (0-9) ***S*** = +, - or nothing
(*uf***)**	Grouping intermediate computations.
-*uf*	Negate *uf*
uf0 **+** *uf1*	Add *uf0* and *uf1*
uf0 **-** *uf1*	Subtract *uf1* from *uf0*
uf0 ***** *uf1*	Multiply *uf0* and *uf1*
uf0 **/** *uf1*	Divide *uf0* by *uf1*
*uc***[***uf0***]**	Gives channel *floor(uf0)* of color *uc*, where red is channel 0, green is channel 1, blue is channel 2 and alpha is channel 3.
*um***[***uf0***] [***uf1***]**	Gives element *floor(4*uf0 + uf1)* of matrix *um*
*ufa***[***uf***]**	Element *floor(uf)* of array *ufa* where element 0 is the first element. Behavior is undefined if *floor(uf0)* falls outside the array.
*f***(...)**	Function call to a function returning uniform float result

Uniform float assignments take the following forms, where *lvalue* is either a uniform float variable or a floating point element from a variable (*var[uf0]* for a uniform color or a uniform float array, *var[uf0][uf1]* for a uniform matrix or uniform color array or *var[uf0][uf1][uf2]* for a uniform matrix array):

lvalue **=** *uf*	Simple assignment
lvalue **+=** *uf*	Equivalent to *lvalue = lvalue + uf*
lvalue **-=** *uf*	Equivalent to *lvalue = lvalue – uf*

lvalue ***=** *uf*	Equivalent to *lvalue = lvalue * uf*
lvalue **/=** *uf*	Equivalent to *lvalue = lvalue / uf*

B. uniform color

Operations producing a uniform color:

variable reference	Value of uniform color variable
color(*uf0***,***uf1***,***uf2***,***uf3***)**	*red=uf0*; *green=uf1*; *blue=uf2*; *alpha=uf3*
uf	*color(uf,uf,uf,uf)*
(*uc***)**	Grouping intermediate computations
-*uc* *uc0* **+** *uc1* *uc0* **-** *uc1* *uc0* ***** *uc1* *uc0* **/** *uc1*	Each uniform float operation is applied component-by-component
*um***[***uf***]**	Row *floor(uf)* of matrix *um*
*uca***[***uf***]**	Element *floor(uf)* of array *uca*, where element 0 is the first element. Behavior is undefined if *floor(uf0)* falls outside the array.
*f***(...)**	Function call to a function returning uniform color result

Uniform color assignments take the following forms, where *lvalue* is either a uniform color variable or a color element from a variable (*var[uf0]* for an element of a color array or row of a uniform matrix or *var[uf0][uf1]* for a uniform matrix array):

lvalue **=** *uc*	Simple assignment
lvalue **+=** *uc*	Equivalent to *lvalue = lvalue + uc*
lvalue **-=** *uc*	Equivalent to *lvalue = lvalue – uc*
lvalue ***=** *uc*	Equivalent to *lvalue = lvalue * uc*
lvalue **/=** *uc*	Equivalent to *lvalue = lvalue / uc*

Color elements can also be set individually. See section A above.

C. uniform matrix

Operations producing a uniform matrix:

variable reference	Value of uniform matrix variable

matrix(*uf0*,*uf1*,*uf2*,*uf3*, *uf4*,*uf5*,*uf6*,*uf7*, *uf8*,*uf9*,*uf10*,*uf11*, *uf12*,*uf13*,*uf14*,*uf15***)**	Matrix with rows *(uf0,uf1,uf2,uf3)*, *(uf4,uf5,uf6,uf7)*, *(uf8,uf9,uf10,uf11)* and *(uf12,uf13,uf14,uf15)*
uf	*matrix(uf,0,0,0, 0,uf,0,0, 0,0,uf,0, 0,0,0,uf)*
(*um***)**	Grouping intermediate computations
-*um* *um0* **+** *um1* *um0* **-** *um1*	Each uniform float operation is applied component-by-component
um0 ***** *um1*	Matrix multiplication: *result[i][k] = sum$_{j=0..3}$(um0[i][j] * um1[j][k])*
*uma***[***uf***]**	Element *floor(uf)* of array *uma* where element 0 is the first element. Behavior is undefined if *floor(uf0)* falls outside the array.
*f***(...)**	Function call to a function returning uniform matrix result

Uniform matrix assignments take the following forms, where lvalue is either a uniform matrix variable or one element of a uniform matrix array variable, accessed as *var[uf]*:

lvalue **=** *um*	Simple assignment
lvalue **+=** *um*	Equivalent to *lvalue = lvalue + um*
lvalue **-=** *um*	Equivalent to *lvalue = lvalue – um*
lvalue ***=** *um*	Equivalent to *lvalue = lvalue * um*

Matrix elements can also be set individually. See sections A and B above.

E. uniform string

Operations producing a uniform string:

variable reference	Value of uniform string variable
constant string	String inside double quotes (*"string"*)
*usa***[***uf***]**	Element *floor(uf)* of array *usa* where element 0 is the first element. Behavior is undefined if *floor(uf0)* falls outside the array.

f`(...)`	Function call to a function returning uniform string result

Strings can include escape sequences beginning with '\':

character sequence	name
O	Octal character code
\x*H*	Hex character code
\n	Newline
\t	Tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\a	Alert (bell)
\\	Backslash character
\?	Question mark
\'	Single quote
\"	Embedded double quote

Uniform string assignments take the following forms, where lvalue is either a uniform string variable or one element of an uniform string array variable, accessed by *var[uf]*:

lvalue **=** *us*	Simple assignment

F. uniform relations

Operations producing a uniform relation (used in control statements discussed later):

uf0 **==** *uf1* *uf0* **!=** *uf1* *uf0* **>=** *uf1* *uf0* **<=** *uf1* *uf0* **>** *uf1* *uf0* **<** *uf1*	Traditional comparisons: equal, not equal, greater or equal, less or equal, greater, and less
uc0 **==** *uc1*	True if all elements of *uc0* are equal to the corresponding elements of *uc1*

uc0 **!=** *uc1*	true if any elements of *uc0* does not equal the corresponding element of *uc1*
um0 **==** *um1*	True if all elements of *um0* are equal to the corresponding elements of *um1*
um0 **!=** *um1*	True if any elements of *um0* does not equal the corresponding element of *um1*
us0 **==** *us1* *us0* **!=** *us1*	Traditional string comparison: equal and not equal
(*ur*)	Grouping intermediate computations
ur0 **&&** *ur1*	True if both *ur0* and *ur1* are true
ur0 **||** *ur1*	True if either *ur0* or *ur1* are true
!*ur*	True if *ur* is false

It is not possible to save uniform relation results to a variable.

VI. Parameter operations

In the following, *pf* and *pf0*- *pf15* are parameter floats; *pfa* is an array of parameter floats; *pc*, *pc0* and *pc1* are parameter colors; *pca* is an array of parameter colors; *pm*, *pm0* and *pm1* are parameter matrices; and *pma* is an array of parameter matrices. Also, *uf0* and *uf1* are uniform floats and *uc* is a uniform color as defined above.

A. parameter float

Operations producing a parameter float:

variable reference	Value of parameter float variable.
uf	Convert uniform float to parameter float.
(*pf*)	Grouping intermediate computations.
-*pf*	Negate *pf*
pf0 **+** *pf1*	Add *pf0* and *pf1*
pf0 **-** *pf1*	Subtract *pf1* from *pf0*
pf0 ***** *pf1*	Multiply *pf0* and *pf1*
pf0 **/** *pf1*	Divide *pf0* by *pf1*
pc[*pf0*]	Gives channel *floor(pf0)* of color *pc*, where red is channel 0, green is channel 1, blue is channel 2 and alpha is channel 3.
pm[*pf0*] [*pf1*]	Gives element *floor(4*pf0 + pf1)* of matrix *pm*

pfa[*uf*]	Element *floor(uf)* of array *pfa* where element 0 is the first element. Note that currently the array index must be uniform. Behavior is undefined if *floor(uf0)* falls outside the array.
f(...)	Function call to a function returning parameter float result

Parameter float assignments take the following forms, where *lvalue* is either a parameter float variable or a floating point element from a variable (*var[uf0]* for a parameter float array):

lvalue = *pf*	Simple assignment
lvalue += *pf*	Equivalent to *lvalue = lvalue + pf*
lvalue -= *pf*	Equivalent to *lvalue = lvalue - pf*
lvalue *= *pf*	Equivalent to *lvalue = lvalue * pf*
lvalue /= *pf*	Equivalent to *lvalue = lvalue / pf*

B. parameter color

Operations producing a parameter color:

variable reference	Value of parameter color variable
uc	Convert uniform color to parameter color.
color(*pf0*,*pf1*,*pf2*,*pf3*)	*red=pf0*; *green=pf1*; *blue=pf2*; *alpha=pf3*
pf	*color(pf,pf,pf,pf)*
(*pc*)	Grouping intermediate computations
-*pc* *pc0* + *pc1* *pc0* - *pc1* *pc0* * *pc1* *pc0* / *pc1*	Each parameter float operation is applied component-by-component
pm[*pf*]	Row *floor(pf)* of matrix *pm*
pca[*uf*]	Element *floor(uf)* of array *pca*, where element 0 is the first element. Note that currently the array index must be uniform. Behavior is undefined if *floor(uf0)* falls outside the array.
f(...)	Function call to a function returning parameter color result

Parameter color assignments take the following forms, where *lvalue* is either a parameter color variable or a color element from a variable (*var[uf0]* for an element of

a color array):

lvalue **=** *pc*	Simple assignment
lvalue **+=** *pc*	Equivalent to *lvalue = lvalue + pc*
lvalue **-=** *pc*	Equivalent to *lvalue = lvalue − pc*
lvalue ***=** *pc*	Equivalent to *lvalue = lvalue * pc*
lvalue **/=** *pc*	Equivalent to *lvalue = lvalue / pc*

Unlike uniform colors, parameter colors cannot currently be set by element.

C. parameter matrix

Operations producing a parameter matrix:

variable reference	Value of parameter matrix variable
um	Convert uniform matrix to parameter matrix.
matrix(*pf0***,***pf1***,***pf2***,***pf3***,** *pf4***,***pf5***,***pf6***,***pf7***,** *pf8***,***pf9***,***pf10***,***pf11***,** *pf12***,***pf13***,***pf14***,***pf15***)**	Matrix with rows *(pf0,pf1,pf2,pf3)*, *(pf4,pf5,pf6,pf7)*, *(pf8,pf9,pf10,pf11)* and *(pf12,pf13,pf14,pf15)*
pf	*matrix(pf,0,0,0, 0,pf,0,0, 0,0,pf,0, 0,0,0,pf)*
(*pm***)**	Grouping intermediate computations
-*pm* *pm0* **+** *pm1* *pm0* **-** *pm1*	Each parameter float operation is applied component-by-component
pm0 ***** *pm1*	Matrix multiplication: *result[i][k] = sum$_{j=0..3}$(um0[i][j] * um1[j][k])*
*pma***[***uf***]**	Element *floor(uf)* of array *pma* where element 0 is the first element. Note that currently the array index must be uniform. Behavior is undefined if *floor(uf0)* falls outside the array.
*f***(...)**	Function call to a function returning parameter matrix result

Parameter matrix assignments take the following forms, where lvalue is either a parameter matrix variable or one element of a parameter matrix array variable, accessed as *var[uf]*:

lvalue **=** *pm*	Simple assignment

lvalue **+=** *pm*	Equivalent to *lvalue = lvalue + pm*
lvalue **-=** *pm*	Equivalent to *lvalue = lvalue – pm*
lvalue ***=** *pm*	Equivalent to *lvalue = lvalue * pm*

Unlike uniform matrices, parameter matrices cannot currently be set by element.

D. Parameter relations

Operations producing a parameter relation closely parallel the uniform relations covered earlier. They can be used in control statements discussed later:

pf0 **==** *pf1* *pf0* **!=** *pf1* *pf0* **>=** *pf1* *pf0* **<=** *pf1* *pf0* **>** *pf1* *pf0* **<** *pf1*	Traditional comparisons: equal, not equal, greater or equal, less or equal, greater, and less
pc0 **==** *pc1*	True if all elements of *pc0* are equal to the corresponding elements of *pc1*
pc0 **!=** *pc1*	true if any elements of *pc0* does not equal the corresponding element of *pc1*
pm0 **==** *pm1*	True if all elements of *pm0* are equal to the corresponding elements of *pm1*
pm0 **!=** *pm1*	True if any elements of *pm0* does not equal the corresponding element of *pm1*
(*pr***)**	Grouping intermediate computations
pr0 **&&** *pr1*	True if both *pr0* and *pr1* are true
pr0 **||** *pr1*	True if either *pr0* or *pr1* are true
!*pr*	True if *pr* is false

It is not possible to save parameter relation results to a variable.

VII. Varying operations

In the following, *vc* is a varying color. Also, *pf0* and *pf1* are parameter floats and *pc* is a parameter color as defined above.

A. varying color

Operations producing a varying color:

variable reference	Value of varying color variable Note: when a varying variable is used, *texgen* value of -3 is passed to the application geometry drawing function (see the description under *texture()*). While the geometry drawing function may choose to act on this value, OpenGL Shader will set the texture generation mode appropriately.
pc	Convert parameter color to varying, clamping the resulting color to [0,1]. After this conversion, every pixel has its own copy of the color value.

Possible targets for varying assignments are:

FB	All channels of the framebuffer
FB.*C*	Set only some channels, leaving the others alone. *C* is a channel specification, consisting of some combination of the letters *r*,*g*,*b* and *a* to select the red, green, blue and alpha channels. Each letter can appear at most once, and they must appear in order. This can be used to isolate individual channels: *FB.r*, *FB.g*, *FB.b*, *FB.a*, or to select arbitrary groups of channels: *FB.rgb*, *FB.rb*, *FB.ga*.

Varying assignments into the framebuffer can take the following forms, where *lvalue* is *FB* or *FB.C* (as described above):

FB = *f*(...)	Function call to a function returning varying color result All varying functions also implicitly have access to the value of FB when the function is called. Except for certain built-in functions explicitly noted later, varying functions can **only** be assigned directly into all channels of the framebuffer. To combine the results of a varying function with the existing frame buffer contents, you must save the existing frame buffer into a variable. For example:

NO	OK

`FB.r = f();`	`varying color a = FB;`
	`FB = f();`
	`FB.bga = a;`

lvalue = *vc*	Copy *vc* into *lvalue*

lvalue += *vc* *lvalue* -= *vc* *lvalue* *= *vc*	Add, subtract, or multiply *lvalue* and *vc*, putting the result in *lvalue*.

Assignments into varying variables can only take this form:

variable =	**FB**	Copy framebuffer to variable

B. varying relations

Operations producing a varying relation (used in control statements discussed later):

FB[*vf0***] == *vf1* **FB[***vf0***] != *vf1* **FB[***vf0***] >= *vf1* **FB[***vf0***] <= *vf1* **FB[***vf0***] > *vf1* **FB[***vf0***] < *vf1*	Traditional comparisons: equal, not equal, greater or equal, less or equal, greater, and less Performs per-pixel comparison between frame buffer channel *uf0* and reference value *uf1*. Frame buffer channel 0 is red, channel 1 is green, channel 2 is blue and channel 3 is alpha.

It is not possible to save varying relation results to a variable.

VIII. Built-in functions

The following is the set of provided functions returning uniform results.

uniform float abs(uniform float *x***)** **parameter float abs(parameter float** *x***)**	absolute value of x
uniform float acos(uniform float *x***)** **parameter float acos(parameter float** *x***)**	inverse cosine, radian result is between 0 and pi
uniform float asin(uniform float *y***)**	inverse sine, radian result is between -pi/2 and pi/2

parameter float asin(parameter float *y***)**	
uniform float atan(uniform float *f***)** **parameter float atan(parameter float** *f***)**	inverse tangent, radian result is between -pi/2 and pi/2
uniform float atan(uniform float *y***; uniform float** *x***)** **parameter float atan(parameter float** *y***; parameter float** *x***)**	inverse tangent of y/x, radian result is between -pi and pi
uniform float ceil(uniform float *x***)** **parameter float ceil(parameter float** *x***)**	round x up (smallest integer i >= x)
uniform float clamp(uniform float *x***; uniform float** *a***; uniform float** *b***)** **parameter float clamp(parameter float** *x***; parameter float** *a***; parameter float** *b***)**	clamp x to lie between a and b
uniform float cos(uniform float *r***)** **parameter float cos(parameter float** *r***)**	cosine of *r* radians
uniform float exp(uniform float *x***)** **parameter float exp(parameter float** *x***)**	e^x
uniform float floor(uniform float *x***)** **parameter float floor(parameter float** *x***)**	round x down (largest integer i <= x)
uniform matrix inverse(uniform matrix *m***)**	matrix inverse *m*inverse(m) = inverse(m)*m =* identity matrix

parameter matrix inverse(parameter matrix *m*)	
uniform float log(uniform float *x*) **parameter float log(parameter float** *x*)	natural log of x
uniform float max(uniform float *x*; **uniform float** *y*) **parameter float max(parameter float** *x*; **parameter float** *y*)	maximum of x and y
uniform float min(uniform float *f*; **uniform float** *g*) **parameter float min(parameter float** *f*; **parameter float** *g*)	minimum of x and y
uniform float mod(uniform float *n*; **uniform float** *d*) **parameter float mod(parameter float** *n*; **parameter float** *d*)	Remainder of division *n/d* *n - d*floor(n/d)*
uniform matrix perspective(uniform float *d*) **parameter matrix perspective(parameter float** *d*)	matrix to perform perspective projection looking down the Z axis with a field of view of *d* degrees. *matrix(cotan(d/2),0, 0, 0,* *0, cotan(d/2),0, 0,* *0, 0, 1, 1,* *0, 0, -2,0)*
uniform float pow(uniform float *x*; **uniform float** *y*) **parameter float pow(parameter float** *x*; **parameter float** *y*)	x^y
uniform matrix rotate(uniform float *x*; **uniform float** *y*; **uniform float** *z*; **uniform float** *r*)	rotate *r* radians around axis *(x,y,z)*

parameter matrix **rotate(parameter float** *x***;** **parameter float** *y***;** **parameter float** *z***;** **parameter float** *r***)**	
uniform float round(uniform **float** *x***)** **parameter float** **round(parameter float** *x***)**	round x to the nearest integer
uniform matrix scale(uniform **float** *x***; uniform float** *y***;** **uniform float** *z***)** **parameter matrix** **scale(parameter float** *x***;** **parameter float** *y***;** **parameter float** *z***)**	*matrix(x,0,0,0, 0,y,0,0, 0,0,z,0, 0,0,0,1)*
uniform float sign(uniform **float** *x***)** **parameter float** **sign(parameter float** *x***)**	sign of x: -1, 0 or 1
uniform float sin(uniform **float** *r***)** **parameter float** **sin(parameter float** *r***)**	sine of *r* radians
uniform float **smoothstep(uniform float** *a***; uniform float** *b***;** **uniform float** *x***)** **parameter float** **smoothstep(parameter** **float** *a***; parameter float** *b***; parameter float** *x***)**	smooth transition between 0 and 1 as *x* changes from *a* to *b*. 0 for *x* < *a*, 1 for *x* > *b*
uniform color spline(uniform **float** *x***; uniform color** *c***[])** **uniform float spline(uniform** **float** *x***; uniform float** *c***[])**	evaluate Catmull-Rom spline at *x* based on control point vector, *c*. A Catmull-Rom spline passes through all of the control points. The derivative of the curve at each control point is half the difference between the next and previous control points. The full curve is covered between *x=0* and *x=1*

parameter color **spline(parameter float** *x***;** **parameter color** *c***[])** **parameter float** **spline(parameter float** *x***;** **parameter float** *c***[])**	
uniform float sqrt(uniform **float** *x***)** **parameter float** **sqrt(parameter float** *x***)**	square root of *x*
uniform float step(uniform **float** *a***; uniform float** *x***)** **parameter float** **step(parameter float** *a***;** **parameter float** *x***)**	0 for *x<a* 1 for *x>=a*
uniform float tan(uniform **float** *r***)** **parameter float** **tan(parameter float** *r***)**	tangent of *r* radians
uniform matrix **translate(uniform float** *x***;** **uniform float** *y***; uniform** **float** *z***)** **parameter matrix** **translate(parameter float** *x***; parameter float** *y***;** **parameter float** *z***)**	*matrix(1,0,0,0, 0,1,0,0, 0,0,1,0,* *x,y,z,1)*

The following is the set of provided functions returning varying color results.

varying color texture(**uniform string** *texturename[***;** **parameter matrix** *xform[***;** **uniform float** *texgen]]***)** **varying color texture(** **uniform float**	Map texture onto surface, using texture coordinates defined with object geometry. Versions with array textures are 1D texturing only (using the *s* texture coordinate). Optional float *texgen* (>= 0) is passed to the geometry drawing function so it can generate a different (application defined) set of per-vertex texture coordinates. If *texgen* is not given, a value of 0 will be passed to the geometry drawing function.

<table>
<tr>
<td><pre>
   texturearray[][;
   parameter matrix
   xform[;
   uniform float
   texgen]])

varying color texture(
   uniform color
   texturearray[][;
   parameter matrix
   xform[;
   uniform float
   texgen]])
</pre></td>
<td>Optional matrix xform is a matrix for transforming the texture coordinates. If xform is not given, the identity matrix is used (i.e. texture coordinates are used as given).

Note: negative texgen values are used for built-in texture generation modes. These negative values are also passed to the geometry drawing function. While the geometry drawing function may choose to act on these value, OpenGL Shader will set the texture generation mode appropriately.
<table>
<tr><th>texture use</th><th>texgen code</th></tr>
<tr><td>texture()</td><td>>= 0</td></tr>
<tr><td>project()</td><td>-1</td></tr>
<tr><td>environment()</td><td>-2</td></tr>
<tr><td>varying variable use</td><td>-3</td></tr>
</table>
</td>
</tr>
<tr>
<td><pre>
varying color
   environment(
   uniform string
   texturename[;
   parameter matrix
   xform])

varying color
   environment(
   uniform float
   texturearray[][;
   parameter matrix
   xform])

varying color
   environment(
   uniform color
   texturearray[][;
   parameter matrix
   xform])
</pre></td>
<td>Map texture onto surface, as a spherical environment map. Versions with array textures are 1D texturing only (using the s texture coordinate).

Optional matrix xform is a matrix for transforming the texture coordinates. For example, it can be used to set the map up direction. If xform is not given, the identity matrix is used (i.e. texture coordinates are used as generated).

Note: environment also passes a texgen value of -2 to the application geometry drawing function.</td>
</tr>
<tr>
<td><pre>
varying color project(
   uniform string
   texturename[;
   parameter matrix
   xform])

varying color project(
   uniform float
   texturearray[][;
</pre></td>
<td>Project texture onto surface using parallel projection down the Z axis. Versions with array textures are 1D texturing only (using the X coordinate only).

Optional matrix xform is a matrix for transforming before projection. For example, to project in shader space, use inverse(shadermatrix). If xform is not given, the identity matrix is used.</td>
</tr>
</table>

parameter matrix *xform]*) **varying color project(** **uniform color** *texturearray*[]*[*; **parameter matrix** *xform]*)	Note: *project()* also passes a *texgen* value of -1 to the application geometry drawing function.
varying color transform(parameter matrix *xform*)	Transform the varying color in the frame buffer by the given matrix
varying color lookup(parameter float *lut*[]) **varying color lookup(parameter color** *lut*[])	Lookup each frame buffer channel in the given lookup table. Each channel is handled independently, so the resulting red component of the result comes from the red component *lut[n*FB.r]*. Similarly, for green from *lut[n*FB.g]* and blue from *lut[n*FB.b]*
varying color blend(varying color *v*)	Channel by channel blend: *FB*(1−v) + v = v*(1−FB) + FB*
varying color over(varying color *v*)	Alpha-based blend of *FB* over *v*: *v*(1−FB.a) + FB*FB.a*
varying color under(varying color *v*)	Alpha-based blend of *FB* under *v*: *FB*(1−v.a) + v*v.a*
varying color setupLight(**parameter float** *lightnum*)	Configure a specific light for subsequent diffuse or specular calculations. After being called, the global *lightVector* is set with the current light's position. Light shaders can modify *lightVector* within their body
varying color ambient()	Return sum of ambient light hitting surface
varying color ambient(**uniform float** *lightnum*)	Return result of ambient light *lightnum* If *lightnum*<0 or *lightnum*>=*numambientlights*, ambient() returns black
varying color diffuse()	Return sum of diffuse light hitting surface
varying color diffuse(**uniform float** *lightnum*)	Return result of diffuse contribution from light *lightnum* If *lightnum*<0 or *lightnum*>=*numdirectlights*, diffuse() returns black diffuse(*lightnum*) is equivalent to

	setupLight(lightnum); *runDiffuse(lightVector);*
varying color runDiffuse(parameter color *lvector* **)**	Calculate diffuse effects of previously configured light (configured by using *setupLight*). Accepts a parameter *lvector* to specifiy the light position. Use the global *lightVector* to accept the value set by previous code or the *setupLight* routine. test
varying color specular(parameter float *e***)**	Return sum of specular light hitting surface, using *e* as the exponent in the Phong lighting model
varying color specular(uniform float *lightnum*, **parameter float** *e***)**	Return result of specular contribution from light *lightnum* If *lightnum*<0 or *lightnum*>=*numdirectlights*, specular() returns black specular(*lightnum*, *e*) is equivalent to *setupLight(lightnum);* *runSpecular(e,lightVector);*
varying color runSpecular(parameter float *e***; parameter color** *lvector* **)**	Calculate specular effects of previously configured light (configured by using *setupLight*). Accepts the parameter *e* as the exponent in the Phong lighting model.Accepts a parameter *lvector* to specifiy the light position. Use the global *lightVector* to accept the value set by previous code or the *setupLight* routine.

IX. Variable declarations

A variable declaration is a type name followed by one (and only one) variable name. Each variable name may optionally be followed by an initial value. Some examples:

```
uniform float fvar;
uniform float farray[3];
uniform float fvar = 3;
parameter matrix = 1;
uniform string = "mytexture"
varying color cvar;
```

Variable and functions names are bound using static scoping rules similar to *C*. The same name cannot occur more than once within the same block of statements (bounded by '*{*' and '*}*'), but can be redefined within a nested block:

not legal	legal

{
  uniform float x;
  uniform float x;
}
	{
	  uniform float x;
	  {
	  uniform color x;
	  }
	}

X. Statements

In the following, *uf* is a uniform float, *ur* is a uniform relation and *vr* is a varying relation as defined above.

Legal ISL statements are:

assignment;	Performs assignment
variable declaration;	Creates and possibly initializes variable
{*list of 0 or more statements*}	Executes statements sequentially
if (*ur*) *statement* **if** (*pr*) *statement*	Execute statement only if uniform relation *ur* or parameter relation *pr* is true
if (*ur*) *statement* **else** *statement* **if** (*pr*) *statement* **else** *statement*	Execute first statement if *ur* or *pr* is true, and second statement if *ur* or *pr* is false.
if (*vr*) *statement*	Restricts the currently active set of pixels to those where the given varying relation is true. The active set of pixels starts as all visible pixels within the shaded object, but may be restricted by one or more *if* statements. Note: Any variable of any type assigned inside a varying *if* should only be used inside the *if*. The contents outside the *if* are undefined, and may change from release to release. Assignments into FB are still OK.
if (*vr*) *statement* **else** *statement*	The first statement executes with the same restricted set of pixels as the previous *if* statement. The second statement executes with the active pixels restricted to those that were active when the *if* statement was reached but where the varying relation was false. Note: Any variable of any type assigned inside a varying *if* should only be used inside the *if*. The contents outside the *if* are undefined, and may change from release to release. Assignments into FB are still OK.

repeat (*uf*)
  statement
 	
repeat (*pf*)
  statement
	repeat statement *max(0,floor(uf))* or *max(0,floor(pf))* times.

XI. Functions

Every function has this form:
type function_name(formal_parameters) { body }

The type is one of the ordinary types or a shader type:

surface	Surface appearance. Should compute the base surface color and lighting contribution (though calls to *ambient()*, *diffuse()* and *specular()*).
atmosphere	Equivalent to surface. Atmospheric effects like fog are handled in the last surface shader in the shader list.
ambientlight	Light contributing to *ambient()* function.
distantlight **pointlight**	*distantlight* is a light shining down the z axis. It is transformed by *shadermatrix*, which can be used by the application to point the light in other directions. Within the body of a *distantlight*, *lightVector* gives the light direction. It is initialized to *shadermatrix[2]*, but can be changed by the shader. *pointlight* is a light positioned at the origin. It is transformed by *shadermatrix*, which can be used by the application to point the light in other directions. Within the body of a *pointlight*, *lightVector* gives the light direction. It is initialized to *shadermatrix[3]*, but can be changed by the shader. Distant and point lights return the varying color and intensity of light falling on a surface. They do not compute the interaction of light with the surface itself, that interaction is computed in the surface shader through the *diffuse()* and *specular()* functions, or through *setupLight()* and *runDiffuse()* and *runSpecular*

The set of formal parameter declarations are a semi-colon separated list of uniform variable declarations, with initial values. *Initial values are required for all formal parameters*. For shaders, the initial values are interpreted as defaults for any variable not set explicitly by the application. Arrays in the formal parameter list for a shader are not currently visible to the application. The initial values for parameters of ordinary functions are not currently used, but they are still required.

The body is just a list of statements. The result of each shader is just the value left in *FB* when the shader exits.

The last statement of any function should be the special statement **return** *value;*.

The *return* statement can only appear as the last statement in a function, and the type of *value* should match the function type. For functions returning a varying color, the *return* is optional. If *return* is omitted on a varying color function, the function return value is the value of *FB* at the end of the function.

Surface shaders return a varying color giving the final color of the surface. At the start of the shader, *FB* contains the color of the closest surface previously seen at each pixel. Shaders with transparency should handle any blending with this existing color. In order for surfaces with varying opacity to work, it is also necessary that the application and/or scene graph sort transparent surfaces, and surfaces with varying opacity should be treated as transparent.

Atmosphere shaders start with *FB* set to the final rendered color for each pixel. They return the attenuated color.

An example shader:

```
surface shadertest(
    uniform color c = color(1,0,0,1);
    uniform float f = .25)
{
    FB = diffuse();
    FB *= c*f;
    return FB;
}
```

Chapter 8

Complex Single and Multi-Pass Shading

Bill Mark

Stanford Real-Time Procedural Shading System

SIGGRAPH 2002 Course Notes

William R. Mark, April 4, 2002

The Stanford real-time procedural shading system compiles shaders written in a high-level shading language to graphics hardware. In particular, the system can compile to graphics hardware with programmable vertex and fragment pipelines.

Some of the key features of the system are:

- The user writes shaders in a high-level, hardware-independent shading language.
- The shading language supports multiple computation frequencies. These computation frequencies – fragment, vertex, and primitive-group – map well to graphics hardware.
- The system uses a well-defined internal interface to support a variety of compiler back ends. A different compiler back end can be used for each computation frequency. Each compiler back end targets a particular hardware interface (e.g. register-combiner fragment hardware).
- The system includes compiler back ends that target programmable vertex and fragment hardware.

We have written two papers that discuss various aspects of our system:

- A Real-Time Procedural Shading System for Programmable Graphics Hardware. Kekoa Proudfoot, William R. Mark, Zvetoslav Tzvetkov, Pat Hanrahan. SIGGRAPH 2001. *This paper describes the complete system.*
- Compiling to a VLIW Fragment Pipeline. William R. Mark and Kekoa Proudfoot. SIGGRAPH/Eurographics Graphics Hardware 2001.
  This paper describes the system's compiler for the register-combiner architecture.

The material in these course notes complements these publications. We have included the following:

1. An example shader (our bowling-pin shader), and the compiled code that our system produces for that shader. The compiled code is for a GeForce3 – it includes fragment code (register-combiner configuration), vertex code (NV_vertex_program code), and primitive-group code (X86 CPU code).
2. Documentation for our system's immediate-mode interface. This interface is used to specify and compile shaders; to specify geometry to be rendered; and to set shader parameters. This interface is a layer that runs on top of OpenGL.
3. An example program that uses our system's immediate-mode interface.
4. Documentation for our system's shading language, with a variety of example shaders.

Additional information is available on our project web page,
`http://graphics.stanford.edu/projects/shading`.

Bowling-Pin Shader and Functions Called by It

Bowling-Pin Surface Shader

```
//
// This shader does the complete bowling pin, and fits into a single pass
// on the GeForce3
//
surface shader float4
bowling_pin(texref basemarks, texref decals, texref bumps, float4 uv) {

    // Compute texture coordinates
    float4  uv_wrap  = { uv[0], 10 * Pobj[1], 0, 1 };
    float4  uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
    matrix4 t_basemarks  = invert(translate(2.0, -7.5, 0) * scale (4,  15, 1));
    float4  uv_basemarks = t_basemarks * uv_wrap;
    float4  uv_bumps     = uv_basemarks;
    matrix4 t_decals  = scale(0.5, 1, 1) *
                        invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
    float4  uv_front  = t_decals * uv_label;
    float4  uv_back   = {1.0 - uv_front[0], uv_front[1], uv_front[2], 1};
    float   front     = select(Pobj[2] >= 0, 1, 0) * select(uv[0] > 3, 0, 1);
    float4  uv_decals = select(front==1, uv_front, uv_back);

    // Look up textures
    float4 Decals    = texture(decals,    uv_decals);
    float4 BaseMarks = texture(basemarks, uv_basemarks);
    float  Marks     = alpha(BaseMarks);
    float3 Base      = rgb(BaseMarks);

    // Compute color, primarily by calling separate 'lightmodel_bumps' routine
    float3 Ma = {.4,.4,.4};
    float3 Md = {.5,.5,.5};
    float3 Ms = {.3,.3,.3};
    float3 Kd = rgb((Decals over {Base, 1.0}) * Marks);
    float3 C = lightmodel_bumps(Kd * Ma, Kd * Md, Ms, bumps, uv_bumps);
    return {C, 1.0};
}
```

Light Shader

(the compiled code given later includes one instance of 'simple_light')

```
// helper function for light shader
light float atten (float ac, float al, float aq) {
    return 1.0 / (aq * Sdist * Sdist + al * Sdist + ac);
}

light shader float4 simple_light (float4 color, float ac, float al, float aq) {
    return color * atten(ac, al, aq);
}
```

Bump-map Function Called by Bowling-pin Shader

```
surface float3
lightmodel_bumps(float3 a, float3 d, float3 s, texref bumps, floatv uv_bumps) {

  // Compute normalized tangent-space light vectors
  vertex perlight float3 Ltan = tangentspace(L);
  vertex perlight float3 Htan = tangentspace(H);

  // Lookup from bump map
  float4 Nlookup  = texture(bumps, uv_bumps); // alpha has short len
  float3 Nbump    = 2.0*(rgb(Nlookup)-triple(0.5));
  float  N_avglen = Nlookup[3]; // Length of mipmap filtered N, before renorm

  // Diffuse
  //perlight float3 Lfrag = 2.0*(cubenorm(Ltan)-{.5,.5,.5});
  perlight float3 Lfrag  = Ltan; // Interpolate
  perlight float  NdotL  = dot(Nbump, Lfrag);
  perlight float  shadow = 4*(Lfrag[2] + Lfrag[2]);  // Geometric shadow ramp
  perlight float3 diff   = d * clamp01(NdotL) * clamp01(shadow) * N_avglen;

  // Specular
  perlight float3 Hnorm   = normalize((fragment perlight float3) Htan);
  perlight float  NdotH   = clamp01(dot(Nbump, Hnorm));
  perlight float  NdotHs  = select(Hnorm[2] >= 0, NdotH, 0.0);
  perlight float  NdotH2  = NdotHs * NdotHs;
  perlight float  NdotH4  = NdotH2 * NdotH2;
  perlight float  NdotH8  = NdotH4 * NdotH4;
  perlight float3 spec    = NdotH8 * shadow * s;

  // Combine
  perlight float3 C = diff + spec;
  return integrate(rgb(Cl) * C) + a;
} // lightmodel_bumps
```

Other Functions Called by Bowling-pin Shader

```
surface float3
tangentspace(float3 V) {
  // Convert vector to tangent space, and normalize it
  float VtanX = dot(V,T);
  float VtanY = dot(V,B);
  float VtanZ = dot(V,N);
  return normalize({VtanX, VtanY, VtanZ});
}

// Clamp scalar to range [0,1]
surface clampf clamp01(float x) {return (clampf) x;}
```

Compiler-Generated Fragment Code for Bowling-Pin Shader (Register Combiner Configuration)

```
CLAMPING NOTATION: [] = clamp to [0,1].  {} = clamp to [-1,1]

*** TEXTURE SHADER CONFIG ***
STAGE 0: TEXTURE_2D      TEXREF='decals'      COORD= 'uv_decals'
STAGE 1: TEXTURE_2D      TEXREF='basemarks'   COORD= 'uv_basemarks'
STAGE 2: TEXTURE_2D      TEXREF='bumps'       COORD= 'uv_bumps'
STAGE 3: TEXTURE_CM      TEXREF=CUBENORM      COORD= 'Htan'

***** GLOBAL PASS INPUTS *****
V0.rgb  = interpolate(0.5*(Ltan+{1,1,1}));
V1.rgb  = interpolate(Cl);
T0.rgba = TEXSHADE.rgba
T1.rgba = TEXSHADE.rgba
T2.rgba = TEXSHADE.rgba
T3.rgb  = TEXSHADE.rgb

************ RGB STAGE 0 **************                  ********** ALPHA STAGE 0 *************
T3.rgb = {L}      L = (2*[T2.rgb]-1) dot (2*[T3.rgb]-1)   S0.a = {L}     L = T3.b
T2.rgb = {R}      R = (2*[T2.rgb]-1) dot (2*[V0.rgb]-1)

************ RGB STAGE 1 **************                  ********** ALPHA STAGE 1 *************
                  L = T0.rgb                                             L = [Z0.a]
                  R = T1.rgb * (1-[T0.aaa])                              R = [T3.b]
T0.rgb = {M}      M = L + R                              V0.a = {M}     M = (S0.a < 0.5) ? L : R

************ RGB STAGE 2 **************                  ********** ALPHA STAGE 2 *************
T0.rgb = {L}      L = T0.rgb * T1.aaa                    V0.a = {L}     L = V0.a * V0.a
                                                         T0.a = {R}     R = T2.b

************ RGB STAGE 3 **************                  ********** ALPHA STAGE 3 *************
T1.rgb = {0.5*L}  L = T0.rgb                             V0.a = {L}     L = V0.a * V0.a

************ RGB STAGE 4 **************                  ********** ALPHA STAGE 4 *************
V0.rgb = {L}      L = T1.rgb * [T0.aaa]                                  L = (2*[V0.b]-1)
T1.rgb = {R}      R = V0.aaa * V0.aaa                                    R = (2*[V0.b]-1)
                                                         T1.a = {4*M}   M = L + R

************ RGB STAGE 5 **************                  ********** ALPHA STAGE 5 *************
V0.rgb = {L}      L = V0.rgb * [T1.aaa]                  V0.a = {L}     L = T1.b * T1.a

PER-STAGE PASS INPUTS FOR STAGE 6:
L0.rgb  = {0.300000, 0.300000, 0.300000}
************ RGB STAGE 6 **************                  ********** ALPHA STAGE 6 *************
                  L = V0.rgb * T2.aaa
                  R = V0.aaa * L0.rgb
V0.rgb = {M}      M = L + R

PER-STAGE PASS INPUTS FOR STAGE 7:
L0.rgb  = {0.400000, 0.400000, 0.400000}
************ RGB STAGE 7 **************                  ********** ALPHA STAGE 7 *************
                  L = V1.rgb * V0.rgb
                  R = T0.rgb * L0.rgb
V0.rgb = {M}      M = L + R

********** RGB FINAL STAGE **********                  ********** ALPHA FINAL STAGE **********
OUT.rgb = [V0.rgb]                                     OUT.a = (1-[Z0.a])
```

Compiler-Generated Vertex Code for Bowling-Pin Shader (NV_vertex_program code)

"constant" registers

```
c[0]-c[3]   = __projection * __modelview
c[4]-c[7]   = __modelview
c[8]        = __lightpos              [light position]
c[9]-c[11]  = affine(__modelview)
c[12]-c[14] = transpose(invert(affine(__modelview)))
c[15]       = color                   [light color]
c[16].x     = (__lightpos[3] == 0.0)  [is the light directional?]
c[16].y     = aq                      [light quadratic attenuation factor]
c[16].z     = al                      [light linear attenuation factor]
c[16].w     = ac                      [light constant attenuation factor]
c[17]       = {0.0961539 0 0.25 -0.5}
c[18]       = {0 0.192308 0.538462 1}
c[19]       = {0 0.0666667 0.5 3}
c[20].x     = 10
```

vertex-source registers

```
v[0]: __position
v[1]: __tangent
v[2]: __binormal
v[3]: __normal
v[4]: uv
```

```
!!VP1.0
DP4  o[HPOS].x, c[0], v[0] ;
DP4  o[HPOS].y, c[1], v[0] ;
DP4  o[HPOS].z, c[2], v[0] ;
DP4  o[HPOS].w, c[3], v[0] ;
DP4  R6.x, c[4], v[0] ;
DP4  R6.y, c[5], v[0] ;
DP4  R6.z, c[6], v[0] ;
DP4  R6.w, c[7], v[0] ;
MOV  R2, R6 ;
RCP  R6.x, R6.w ;
MUL  R7, R2, R6.x ;
ADD  R2, c[8], -R7 ;
MAD  R2, c[16].x, -R2, R2 ;
MOV  R6, c[8] ;
MAD  R2, c[16].x, R6, R2 ;
DP3  R6.x, R2, R2 ;
RSQ  R6.x, R6.x ;
MUL  R6, R2, R6.x ;
DP3  R5.x, c[9], v[1] ;
DP3  R5.y, c[10], v[1] ;
DP3  R5.z, c[11], v[1] ;
DP3  R8.x, R5, R5 ;
RSQ  R8.x, R8.x ;
MUL  R5, R5, R8.x ;
DP3  R1.x, R6, R5 ;
DP3  R4.x, c[9], v[2] ;
DP3  R4.y, c[10], v[2] ;
DP3  R4.z, c[11], v[2] ;
DP3  R8.x, R4, R4 ;
RSQ  R8.x, R8.x ;
MUL  R4, R4, R8.x ;
DP3  R1.y, R6, R4 ;
DP3  R3.x, c[12], v[3] ;
DP3  R3.y, c[13], v[3] ;
DP3  R3.z, c[14], v[3] ;
DP3  R8.x, R3, R3 ;
RSQ  R8.x, R8.x ;
MUL  R3, R3, R8.x ;
DP3  R1.z, R6, R3 ;
DP3  R8.x, R1, R1 ;
RSQ  R8.x, R8.x ;
MAD  R1, R1, R8.x, c[18].wwwx ;
MUL  o[COL0].xyz, -c[17].w, R1 ;
DP3  R8.x, R2, R2 ;
RSQ  R1, R8.x ;
DST  R2, R8.x, R1 ;
DP3  R1.x, R2, c[16].wzyy ;
RCP  R1.x, R1.x ;
MUL  R1, c[15], R1.x ;
MOV  o[COL1].xyz, R1 ;
SGE  R1.x, v[0].z, c[17].y ;
MAD  R1.z, R1.x, -c[17].y, c[17].y ;
MAD  R1.z, R1.x, c[18].w, R1.z ;
SLT  R1.y, c[19].w, v[4].x ;
MAD  R1.x, R1.y, -c[18].w, c[18].w ;
MAD  R1.x, R1.y, c[17].y, R1.x ;
MUL  R8.y, R1.z, R1.x ;
SGE  R8.x, R8.y, c[18].w ;
SGE  R8.z, c[18].w, R8.y ;
MIN  R8.z, R8.x, R8.z ;
MUL  R1.xy, c[20].x, v[0].xyxx ;
MOV  R1.z, c[17].y ;
MOV  R1.w, c[18].w ;
DP4  R2.x, c[17].xyyz, R1 ;
DP4  R2.y, c[18].xyxz, R1 ;
DP4  R2.z, c[18].xxwx, R1 ;
DP4  R2.w, c[18].xxxw, R1 ;
ADD  R1.x, c[18].w, -R2.x ;
MOV  R1.yz, R2.yyzy ;
MOV  R1.w, c[18].w ;
MAD  R1, R8.z, -R1, R1 ;
MAD  o[TEX0], R8.z, R2, R1 ;
MOV  R2.x, v[4].x ;
MUL  R2.y, c[20].x, v[0].y ;
MOV  R2.z, c[17].y ;
MOV  R2.w, c[18].w ;
DP4  R1.x, c[17].zyyw, R2 ;
DP4  R1.y, c[19].xyxz, R2 ;
DP4  R1.z, c[18].xxwx, R2 ;
DP4  R1.w, c[18].xxxw, R2 ;
MOV  o[TEX1], R1 ;
MOV  o[TEX2], R1 ;
DP3  R2.x, -R7, -R7 ;
RSQ  R2.x, R2.x ;
MAD  R1, -R7, R2.x, R6 ;
DP3  R2.x, R1, R1 ;
RSQ  R2.x, R2.x ;
MUL  R1, R1, R2.x ;
DP3  R0.x, R1, R5 ;
DP3  R0.y, R1, R4 ;
DP3  R0.z, R1, R3 ;
DP3  R1.x, R0, R0 ;
RSQ  R1.x, R1.x ;
MUL  o[TEX3].xyz, R0, R1.x ;
END
```

Compiler-Generated Primitive-Group Code for Bowling-Pin Shader (x86 CPU code)

```
push    ebp
mov     ebp, esp
sub     esp, 0x0000000c
push    esi
push    edi
push    ebx
fnstcw  [ebp-4h]
fnclex
mov     edi, [ebp+Ch]
mov     ebx, [ebp+8h]
mov     ebx, [ebx+50h]
mov     eax, [ebx]
mov     [edi], eax
mov     eax, [ebx+4h]
mov     [edi+4h], eax
mov     eax, [ebx+8h]
mov     [edi+8h], eax
mov     eax, [ebx+Ch]
mov     [edi+Ch], eax
mov     eax, [ebp+8h]
mov     eax, [eax+38h]
mov     eax, [eax]
mov     [edi+10h], eax
mov     ebx, [ebp+8h]
mov     ebx, [ebx+30h]
mov     eax, [ebx]
mov     [edi+14h], eax
mov     eax, [ebx+4h]
mov     [edi+18h], eax
mov     eax, [ebx+8h]
mov     [edi+1Ch], eax
mov     eax, [ebx+Ch]
mov     [edi+20h], eax
mov     ebx, [ebp+8h]
mov     ebx, [ebx+28h]
mov     eax, [ebx]
mov     [edi+24h], eax
mov     eax, [ebx+4h]
mov     [edi+28h], eax
mov     eax, [ebx+8h]
mov     [edi+2Ch], eax
mov     eax, [ebx+Ch]
mov     [edi+30h], eax
mov     eax, [ebx+10h]
mov     [edi+34h], eax
mov     eax, [ebx+14h]
mov     [edi+38h], eax
mov     eax, [ebx+18h]
mov     [edi+3Ch], eax
mov     eax, [ebx+1Ch]
mov     [edi+40h], eax
mov     eax, [ebx+20h]
mov     [edi+44h], eax
mov     eax, [ebx+24h]
mov     [edi+48h], eax
mov     eax, [ebx+28h]
mov     [edi+4Ch], eax
mov     eax, [ebx+2Ch]
mov     [edi+50h], eax
mov     eax, [ebx+30h]
mov     [edi+54h], eax
mov     eax, [ebx+34h]
mov     [edi+58h], eax
mov     eax, [ebx+38h]
mov     [edi+5Ch], eax
mov     eax, [ebx+3Ch]
mov     [edi+60h], eax
mov     eax, [ebp+8h]
mov     eax, [eax+40h]
mov     eax, [eax]
mov     [edi+64h], eax
mov     eax, [ebp+8h]
mov     eax, [eax+48h]
mov     eax, [eax]
mov     [edi+68h], eax
mov     eax, [ebp+8h]
mov     eax, [eax+68h]
mov     eax, [eax]
mov     [edi+6Ch], eax
mov     eax, [ebp+8h]
mov     eax, [eax+70h]
mov     eax, [eax]
mov     [edi+70h], eax
mov     eax, [ebp+8h]
mov     eax, [eax+60h]
mov     eax, [eax]
mov     [edi+74h], eax
lea     eax, [edi+24h]
push    eax
lea     eax, [edi+78h]
push    eax
mov     [ebp-Ch], 0x0040105a
call    [ebp-Ch]
add     esp, 0x00000008
lea     eax, [edi+78h]
push    eax
lea     eax, [edi+9Ch]
push    eax
mov     [ebp-Ch], 0x00401672
call    [ebp-Ch]
add     esp, 0x00000008
lea     eax, [edi+9Ch]
push    eax
lea     eax, [edi+C0h]
push    eax
mov     [ebp-Ch], 0x0040120d
call    [ebp-Ch]
add     esp, 0x00000008
mov     ebx, [ebp+8h]
mov     ebx, [ebx]
mov     eax, [ebx]
mov     [edi+E4h], eax
mov     eax, [ebx+4h]
mov     [edi+E8h], eax
mov     eax, [ebx+8h]
mov     [edi+ECh], eax
mov     eax, [ebx+Ch]
mov     [edi+F0h], eax
mov     eax, [ebx+10h]
mov     [edi+F4h], eax
mov     eax, [ebx+14h]
mov     [edi+F8h], eax
mov     eax, [ebx+18h]
mov     [edi+FCh], eax
mov     eax, [ebx+1Ch]
mov     [edi+100h], eax
mov     eax, [ebx+20h]
mov     [edi+104h], eax
mov     eax, [ebx+24h]
mov     [edi+108h], eax
mov     eax, [ebx+28h]
mov     [edi+10Ch], eax
mov     eax, [ebx+2Ch]
mov     [edi+110h], eax
mov     eax, [ebx+30h]
mov     [edi+114h], eax
mov     eax, [ebx+34h]
mov     [edi+118h], eax
mov     eax, [ebx+38h]
mov     [edi+11Ch], eax
mov     eax, [ebx+3Ch]
mov     [edi+120h], eax
mov     eax, [edi+20h]
mov     [edi+124h], eax
mov     [edi+128h], 0x00000000
fld     [edi+124h]
fcomp   [edi+128h]
fnstsw  eax
test    eax, 0x00004000
mov     eax, 0x3f800000
jnz     l_0
xor     eax, eax
l_0:
mov     [edi+12Ch], eax
mov     eax, [edi+14h]
mov     [edi+130h], eax
mov     eax, [edi+18h]
mov     [edi+134h], eax
mov     eax, [edi+1Ch]
mov     [edi+138h], eax
lea     eax, [edi+24h]
push    eax
lea     eax, [edi+E4h]
push    eax
lea     eax, [edi+13Ch]
push    eax
mov     [ebp-Ch], 0x004014ba
call    [ebp-Ch]
add     esp, 0x0000000c
mov     ebx, [ebp+10h]
mov     eax, [edi]
mov     [ebx], eax
mov     eax, [edi+4h]
mov     [ebx+4h], eax
mov     eax, [edi+8h]
mov     [ebx+8h], eax
mov     eax, [edi+Ch]
mov     [ebx+Ch], eax
mov     eax, [edi+10h]
mov     [ebx+10h], eax
mov     eax, [edi+24h]
mov     [ebx+14h], eax
mov     eax, [edi+28h]
mov     [ebx+18h], eax
mov     eax, [edi+2Ch]
mov     [ebx+1Ch], eax
mov     eax, [edi+30h]
mov     [ebx+20h], eax
mov     eax, [edi+34h]
mov     [ebx+24h], eax
mov     eax, [edi+38h]
mov     [ebx+28h], eax
mov     eax, [edi+3Ch]
mov     [ebx+2Ch], eax
mov     eax, [edi+40h]
mov     [ebx+30h], eax
mov     eax, [edi+44h]
mov     [ebx+34h], eax
```

```
mov     eax, [edi+48h]
mov     [ebx+38h], eax
mov     eax, [edi+4Ch]
mov     [ebx+3Ch], eax
mov     eax, [edi+50h]
mov     [ebx+40h], eax
mov     eax, [edi+54h]
mov     [ebx+44h], eax
mov     eax, [edi+58h]
mov     [ebx+48h], eax
mov     eax, [edi+5Ch]
mov     [ebx+4Ch], eax
mov     eax, [edi+60h]
mov     [ebx+50h], eax
mov     eax, [edi+64h]
mov     [ebx+54h], eax
mov     eax, [edi+68h]
mov     [ebx+58h], eax
mov     eax, [edi+6Ch]
mov     [ebx+5Ch], eax
mov     eax, [edi+70h]
mov     [ebx+60h], eax
mov     eax, [edi+74h]
mov     [ebx+64h], eax
mov     eax, [edi+78h]
mov     [ebx+68h], eax
mov     eax, [edi+7Ch]
mov     [ebx+6Ch], eax
mov     eax, [edi+80h]
mov     [ebx+70h], eax
mov     eax, [edi+84h]
mov     [ebx+74h], eax
mov     eax, [edi+88h]
mov     [ebx+78h], eax
mov     eax, [edi+8Ch]
mov     [ebx+7Ch], eax
mov     eax, [edi+90h]
mov     [ebx+80h], eax
mov     eax, [edi+94h]
mov     [ebx+84h], eax
mov     eax, [edi+98h]
mov     [ebx+88h], eax
mov     eax, [edi+C0h]
mov     [ebx+8Ch], eax
mov     eax, [edi+C4h]
mov     [ebx+90h], eax
mov     eax, [edi+C8h]
mov     [ebx+94h], eax
mov     eax, [edi+CCh]
mov     [ebx+98h], eax
mov     eax, [edi+D0h]
mov     [ebx+9Ch], eax
mov     eax, [edi+D4h]
mov     [ebx+A0h], eax
mov     eax, [edi+D8h]
mov     [ebx+A4h], eax
mov     eax, [edi+DCh]
mov     [ebx+A8h], eax
mov     eax, [edi+E0h]
mov     [ebx+ACh], eax
mov     eax, [edi+12Ch]
mov     [ebx+B0h], eax
mov     eax, [edi+130h]
mov     [ebx+B4h], eax
mov     eax, [edi+134h]
mov     [ebx+B8h], eax
mov     eax, [edi+138h]
mov     [ebx+BCh], eax
mov     eax, [edi+13Ch]
mov     [ebx+C0h], eax
mov     eax, [edi+140h]
mov     [ebx+C4h], eax
mov     eax, [edi+144h]
mov     [ebx+C8h], eax
mov     eax, [edi+148h]
mov     [ebx+CCh], eax
mov     eax, [edi+14Ch]
mov     [ebx+D0h], eax
mov     eax, [edi+150h]
mov     [ebx+D4h], eax
mov     eax, [edi+154h]
mov     [ebx+D8h], eax
mov     eax, [edi+158h]
mov     [ebx+DCh], eax
mov     eax, [edi+15Ch]
mov     [ebx+E0h], eax
mov     eax, [edi+160h]
mov     [ebx+E4h], eax
mov     eax, [edi+164h]
mov     [ebx+E8h], eax
mov     eax, [edi+168h]
mov     [ebx+ECh], eax
mov     eax, [edi+16Ch]
mov     [ebx+F0h], eax
mov     eax, [edi+170h]
mov     [ebx+F4h], eax
mov     eax, [edi+174h]
mov     [ebx+F8h], eax
mov     eax, [edi+178h]
mov     [ebx+FCh], eax
fldcw   [ebp-4h]
pop     ebx
pop     edi
pop     esi
mov     esp, ebp
pop     ebp
ret
```

Real-Time Shading Language v6

Kekoa Proudfoot Eric Chan

February 28, 2002

Contents

1 Language version history

The version 1 language had lisp-like parenthetical constructs and shaders, expressions of fixed colors, textures, and lit materials. The only data type was a $[0, 1]$ clamped color, and the allowed operators were `add`, `multiply`, and `blend (over)`.

The version 2 language replaced the lisp-like constructs of the version 1 language with ones more like C. The underlying expressions, operators, and data types did not change.

The version 3 language was discussed but never implemented. The intent was to extend the version 2 language to remove the restriction that colors, textures, and lit materials be fixed by making these data types configurable through parameters to shaders. This language version was also to introduce a separation between light shaders and surface shaders.

The version 4 language allowed shaders to be configured using shader parameters and provided a light/surface shader abstraction. It also introduced the concept of multiple computation frequencies, making use of types to manage when and how computations are performed. New vertex and primitive-group processing capabilities were exposed to complement a set of fragment processing capabilities similar to those available in previous language versions.

The version 5 language allowed us to explore compilation to advanced fragment processing pipelines. The new features included three-component vectors, three-by-three matrices, three-vector operations, more fragment operations, operators to assist with compiling to fragment pipelines, and conditional compilation.

The version 6 language is described in this document. It is an extension of the version 5 language that provides additional operators and functions to assist with compiling to advanced vertex and fragment hardware. In particular, this revision adds the following new features:

- Boolean logical operators (Section 2.2.6)
- Derivative operators (Section 2.2.3)
- General index `operator[]` with swizzle (Section 2.2.1)
- Assignment writemasks (Section 2.2.13)
- An operator to access screen-space position and depth value per-fragment (Section 2.2.11)

2 Basics

The general format of our language, as well as our language's declaration and expression syntax, is similar to C. Our language does, however, have a number of notable differences. These include a different set of data types, a number of specialized type modifiers, a slightly different set of operators, and different semantics with regards to function calls and global variables. These differences will become clearer as you proceed through this document.

As with C, our language relies on white space and indenting only to the extent that they separate tokens in the language. White space and indenting are otherwise ignored.

Comments are allowed in our language. These may be denoted using either the C `/* */` syntax or the C++ `//` comment syntax. Identifiers, integers, and floats are all specified as they are in C. Identifiers are case-sensitive.

2.1 Base Data Types

We begin the discussion of our language with a description of its data types.

In our language, data types are composed of a base data type preceeded by an optional list of type modifiers. In this section, we describe the base data types. We leave the discussion of type modifiers for later sections.

Our language supports ten base data types. They are:

`bool`	boolean value
`clampf1`	scalar $[0, 1]$-clamped floating-point value
`clampf3`	3-component $[0, 1]$-clamped floating-point vector
`clampf4`	4-component $[0, 1]$-clamped floating-point vector
`float1`	scalar unclamped floating-point value
`float3`	3-component unclamped floating-point vector
`float4`	4-component unclamped floating-point vector
`matrix3`	3×3 floating point matrix
`matrix4`	4×4 floating point matrix
`texref`	texture reference

Two of these types need further explanation.

- The `bool` type is either `true` or `false`. It has no numerical value.
- The `ftexref` type stores a reference to a texture. Its value corresponds to an OpenGL texture name as specified to `glBindTexture`.

Additionally, note that although the clamped float types are described as floating point, because their ranges are limited to $[0, 1]$, they may be implemented using either fixed- or floating-point.

In addition to the ten base types, we support some additional type names for compatibility with the previous version of the language:

`clampf`	same as `clampf1`
`clampfv`	same as `clampf4`
`float`	same as `float1`
`floatv`	same as `float4`
`matrix`	same as `matrix4`

2.2 Expressions, Operators, and Built-in Functions

The expression syntax of our language is much like that of C, except that we provide a different set of operators and also a core set of built-in functions. In this section, we introduce and describe these operators and functions.

Most operators that we provide have both `float` and `clampf` versions, where the `clampf` versions are defined to clamp their results (but not their intermediate values) to $[0, 1]$. We make special note of operators which either do not have `clampf` versions or do not operate on `float` or `clampf` values at all.

2.2.1 Operators for manipulating scalars and vectors

The `join` operator `{}` assembles scalars into vectors and vectors into matrices. It comes in five versions:

```
{ x, y, z }         // make a 3-vector from scalars x, y, and z
{ x, y, z, w }      // make a 4-vector from scalars x, y, z, and w
{ xyz, w }          // make a 4-vector from 3-vector xyz and scalar w
{ r0, r1, r2 }      // make a 3x3 matrix from 3-vector rows r0, r1, r2
{ r0, r1, r2, r3 }  // make a 4x4 matrix from 4-vector rows r0, r1, r2, r3
```

The index `operator[]` has many uses. It can be used to extract a scalar from a 3- or 4- component vector, or to swizzle the components of a vector. Indexing is zero-based:

```
float3 vec3 = { x, y, z };
float4 vec4 = { x, y, z, w };

vec3[0]        // extract x
vec3[2]        // extract z
vec4[3]        // extract w
vec3[1,2,0]    // returns { y, z, x }
vec3[2,2,2]    // returns { z, z, z }
vec3[2,0,1,1]  // returns { z, x, y, y }
vec4[3,0,2]    // returns { w, x, z }
```

The number of comma-delimited indices given inside the square braces specifies the size of the output. The output must be a scalar, a 3-component vector, or a 4-component vector. The following is illegal because we do not currently support 2-component vectors:

```
{ x, y, z, w }[1,2]   // error: result is a 2-vector
```

Each element given in the index `operator[]` may be in the range $0 \ldots N-1$, where N is the number of components of the operand. For example:

```
{ x, y, z }[3,0,2]     // error: index 3 out of range
{ x, y, z, w }[3,0,2]  // ok: returns { w, x, z }
```

The index `operator[]` can also extract a row from a 3×3 matrix or a 4×4 matrix.

```
{ r0, r1, r2 }[0]      // extract r0 from 3x3 matrix { r0, r1, r2 }
{ r0, r1, r2 }[2]      // extract r2 from 3x3 matrix { r0, r1, r2 }
{ r0, r1, r2, r3 }[3]  // extract r3 from 4x4 matrix { r0, r1, r2, r3 }
```

The `rgb()`, `alpha()`, and `blue()` operators help make compilation to fragment pipelines efficient. However, they remain primarily for compatibility with older versions of the language. Their various forms and equivalent expressions are shown here:

```
rgb({ r, g, b, a })     // extract 3-vector { r, g, b } from 4-vector;
                        // equivalent to { r, g, b, a }[0,1,2]

alpha({ r, g, b, a })   // extract scalar a from 3-vector;
                        // equivalent to { r, g, b, a }[3]

blue({ r, g, b, a })    // extract scalar b from 4-vector
                        // equivalent to { r, g, b, a }[2]

blue({ r, g, b })       // extract scalar b from 3-vector;
                        // equivalent to { r, g, b }[2]

rgb(c)                  // construct 3-vector { c, c, c } from scalar c
                        // equivalent to c[0,0,0]
```

2.2.2 Arithmetic operators

We provide scalar and vector versions of add, multiply, subtract, and divide. For multiply and divide, we also provide versions that operate on one scalar and one vector in either order. Some examples:

```
a + b
a - b
a * b
a / b
{ ax, ay, az } + { bx, by, bz }
{ ax, ay, az } - { bx, by, bz }
{ ax, ay, az } * { bx, by, bz }
{ ax, ay, az } / { bx, by, bz }
a * { bx, by, bz }
a / { bx, by, bz }
{ ax, ay, az } * b
{ ax, ay, az } / b
```

Multiplication of two matrices and multiplication of one matrix (on the left) and one vector (on the right) are also supported. Since we do not support clamped matrices, there are no `clampf` matrix-matrix or matrix-vector multiply operations.

We provide an unclamped floating-point negate operator:

```
- a
```

We do not provide a `clampf` version of the negate operator, since its result would always be zero.

2.2.3 Derivative operators

We provide derivative operators that operate on unclamped scalars, 3-vectors, and 4-vectors. They compute the partial derivatives of an expression with respect to `x` and `y` in screen-space coordinates.

```
dx(expr)    // computes partial derivative of expr (w.r.t. x)
dy(expr)    // computes partial derivative of expr (w.r.t. y)
```

2.2.4 Blending operators

We provide a generic blend operator that operates on clamped and unclamped 4-vectors only. The blend operator is based on the OpenGL blend function and takes the following form:

```
blend ( src_factor, dst_factor )
```

Note this the blend operator is a binary infix operator. The value to the left of the blend is called the source (`src`) and the value to the right of the blend is called the destination (`dst`):

```
src blend(src_factor,dst_factor) dst
```

Such an expression computes:

```
src_factor * src + dst_factor * dst
```

Both `src_factor` and `dst_factor` are placeholders for names chosen from the following list. Each has the value indicated:

Factor Name	Factor Value
ZERO	{ 0, 0, 0, 0 }
ONE	{ 1, 1, 1, 1 }
SRC_COLOR	src
SRC_ALPHA	{ src[3], src[3], src[3], src[3] }
DST_COLOR	dst
DST_ALPHA	{ dst[3], dst[3], dst[3], dst[3] }
ONE_MINUS_SRC_COLOR	{ 1, 1, 1, 1 } - src
ONE_MINUS_SRC_ALPHA	{ 1, 1, 1, 1 } - { src[3], src[3], src[3], src[3] }
ONE_MINUS_DST_COLOR	{ 1, 1, 1, 1 } - dst
ONE_MINUS_DST_ALPHA	{ 1, 1, 1, 1 } - { dst[3], dst[3], dst[3], dst[3] }

We provide two additional blend operators to simplify the specification of common blend operations. The `over` operator composites two values with premultiplied alpha, and is equivalent to `blend (ONE, ONE_MINUS_SRC_ALPHA)`. The `blend_over` operator composites two values where only second value has premultiplied alpha. The first value has non-premultiplied alpha. It is equivalent to `blend(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)`.

2.2.5 Comparison operators

We provide a standard set of comparison operators (==, !=, >, <, >=, <=) for computing boolean values. We also provide a `lthalf()` operator to assist with fragment compilation. The `lthalf()` operator returns true if its operand is less than $\frac{1}{2}$.

2.2.6 Logical operators

We provide the four standard logical operators `AND`, `OR`, `NOT`, `XOR` that operate on boolean values. `NOT` has the highest precedence, followed by `AND`, `XOR`, `OR`.

```
bool a = true;
bool b = false;

a & b    // a AND b        -> false
a | b    // a OR b         -> true
a ^ b    // a XOR b        -> true
~a       // NOT a          -> false
~b & a   // (NOT b) AND a  -> true
```

2.2.7 Conditional `select` operator

Boolean expressions are used with the conditional `select` operator. The `select` operator takes three parameters: a boolean, a value to return if the boolean is `true`, and a value to return if the boolean is `false`. Some examples:

```
select(0 == 0, t, f)        // value is t
select(0 > 1, t, f)         // value is f
select(lthalf(0), t, f)     // value is t
select(lthalf(0.5), t, f)   // value if f
```

2.2.8 Miscellaneous scalar and vector operations

We provide a number of additional operations, including scalar and vector `clamp`, `min`, and `max` operations; vector `dot`, `length`, and `normalize` operations; a 3-vector `reflect` and `cross` operations; `sin`, `cos`, `pow`, and `sqrt`. Some examples:

```
clamp(0.5, 0, 1)                                  // value is 0.5
clamp({ -1, 0, 1, 2 }, 0, 1)                      // value is { 0, 0, 1, 1 }
clamp({ -1, 1, 3 }, { 0, 0, 1 }, { 1, 2, 2})      // value is { 0, 1, 2 }
min({ -1, 1, 2, 3 }, { 1, 0, 1, 4 })              // value is { -1, 0, 1, 4 }
dot({ 0, 1, 2, 3 }, { 4, 5, 6, 7 })               // value is 38
length({ 3, 4, 0 })                               // value is 5
length({ 1, 1, 1 })                               // value is 1.7320...
length({ 1, 1, 1, 1 })                            // value is 2
normalize({ 0, 0, 2 })                            // value is { 0, 0, 1 }
reflect({ 1, 1, 1 }, { 0, 0, 1 })                 // value is { -1, -1, 1 }
reflect({ 1, 0, 0 }, { 0, 1, 0 })                 // value is { 0, 0, 1 }
sin(3.14159)                                      // value is 0
cos(3.14159)                                      // value is -1
pow(10,2)                                         // value is 100
sqrt(2)                                           // value is 1.4142...
```

2.2.9 Matrix operations

We also provide a number of matrix operations:

`affine`	extracts the upper-left 3×3 matrix from a 4×4 matrix
`frustum`	generates a 4×4 frustum projection matrix
`identity`	generates a 4×4 identity matrix
`invert`	inverts a 3×3 or a 4×4 matrix
`lookat`	generates a 4×4 lookat matrix
`ortho`	generates a 4×4 orthographic projection matrix
`rotate`	generates a 4×4 rotation matrix of an angle about an axis
`scale`	generates a 4×4 scale matrix
`translate`	generates a 4×4 translation matrix
`transpose`	transposes a 3×3 or 4×4 matrix
`identity3`	generates a 3×3 identity matrix
`rotate3`	generates a 3×3 rotation matrix
`scale3`	generates a 3×3 scale matrix

The exact parameters needed for each matrix operation are discussed in the operator appendix, Section 9.1.

2.2.10 Texturing operations

A number of texturing and lookup operations are also available:

`cubemap`	perform a cubemap lookup given a `texref` and a 3-vector
`cubenorm`	perform a 3-vector normalization given a 3-vector
`lut`	perform a component-wise fragment `clamp4` table lookup
`texture`	perform a 2d texture lookup given a `texref` and a 3- or 4-vector
`texture3d`	perform a 3d texture lookup given a `texref` and a 3- or 4-vector
`bumpdiff`	perform a diffuse bumpmap operation
`bumpspec`	perform a specular bumpmap operation (requires `bumpdiff`)

The exact parameters needed for each texture and lookup operation are discussed in the operator appendix, Section 9.1.

The `lut` operator performs a component-wise table lookup of fragment value. It uses the OpenGL color lookup table defined using `glPixelMap`. Our intent is to eventually abstract lookup table specification to allow multiple lookup tables, but currently we only support one color lookup table at a time.

The `bumpdiff` and `bumpspec` operators implement bumpmapping as described for NVIDIA hardware by Mark Kilgard. The `bumpdiff` operator computes the diffuse reflection coefficient given a tangent-space

normal map, texture coordinates, and a tangent-space light vector. The `bumpspec` operator computes the specular reflection coefficient given the same normal map and texture coordinates plus the tangent-space half-angle vector. The `bumpdiff` operator leaves a self-shadowing term in alpha which must be used to modulate the `bumpspec` result. The `blend` operator, configured as `blend(ONE, SRC_ALPHA)`, is used to accomplish this.

2.2.11 Accessing screen-space coordinates

The `xyz_screen()` built-in function provides the coordinates and depth value of the current fragment in screenspace:

```
float4 coords = xyz_screen();  // coords[0] = screen-space x position
                               // coords[1] = screen-space y position
                               // coords[2] = depth (z) value
                               // coords[3] = undefined
```

2.2.12 Parentheses

As with C, we support parentheses `()` for grouping expressions to override the default operator precedences.

2.2.13 Assignment and cast operators

Two special operators are the assignment and cast operators. Both are used as they typically are in C. Assignment implies a cast to the type of the value being set. Type conversion is discussed in greater detail in Section 5.

Assignments may be masked. The indices provided in the mask must be unique and appear in ascending order. Some examples:

```
float4 v = { 2, 5, 7, 15 };

v[3] = 0;                   // v is now { 2, 5, 7, 0 };
v[0] = v[1];                // v is now { 5, 5, 7, 0 };
v[0,1,3] = { 1, 1, 1 };     // v is now { 1, 1, 7, 1 };
v[0,0,1] = { 2, 2, 2 };     // error: 0 is repeated
v[0,1,3] = { 1, 1, 1, 1 };  // error: LHS is 3-vector, RHS is 4-vector
v[2,1,3] = { 1, 1, 1 };     // error: mask indices out of order
```

2.2.14 `integrate()`

Finally, we mention the `integrate()` operator, which we discuss in more detail in Section 3 on surface and light shaders.

2.3 Operator Precedence

We define the following binary operator precedences, by group from lowest precedence to highest precedence:

```
=
== !=
> < >= <=
+ -
blend over blend_over
* /
```

All of the binary operators are left associative, except for =, which is right associative.

2.4 Statements

Our language supports three kinds of statements: variable declarations, expression statements, return statements. Empty statements are permitted; these are ignored.

A variable declaration is similar to C, and consists of a type followed by an identifier followed by an optional initializer followed by a semicolon.

```
float1 f1;                      // declare f1
float1 f2 = 1;                  // declare and initialize f2
float4 v1 = { 1, 2, 3, 4 };     // declare and initialize v1
float4 v2 = f1 * v1;            // declare and initialize v2
```

As with C++, variables may be declared anywhere in a basic block.

Expression statements are simply an expression followed by a semicolon:

```
1;                  // valid but useless, eventually optimized away
N = normalize(N);   // normalize N
NdotL = dot(N,L);   // compute dot product of N and L
```

A return statement is used to indicate the final value of a shader or function:

```
return color;
```

2.5 Functions

Our language allows functions to be defined and called mostly like they are in C, with a few exceptions. First, there is no such thing as a `void` function, and therefore all functions must return a value. Second, there is (currently) no such thing as a function declaration for user-defined functions. All user-defined functions must be defined before they may be used. Finally, recursion is forbidden.

All of these differences are due to the way function calls are implemented. All function calls are inlined.

Here is an example.

```
float4 lerp (float4 a, float4 b, float afrac)
{
    return afrac * a + (1 - afrac) * b
}

float4 bilerp (float4 v00, float4 v01, float4 v10 float4 v11,
               float frac0, float frac1)
{
    float4 v0 = lerp(v00, v01, frac0);
    float4 v1 = lerp(v10, v11, frac0);
    return lerp(v0, v1, frac1);
}
```

3 Surface shaders, light shaders, and the `integrate()` operator

Our language borrows the RenderMan concept of separate surface and light shaders to provide orthogonality between these shading operations. Light shaders compute how much light is incident on a surface, while surface shaders compute the amount of light reflected toward the viewer, possibly querying lights to determine and account for the amount of light arriving from each light source.

Surface and light shaders are written as functions are, except that their return types are preceded by the `shader` modifier plus also either the `surface` or the `light` modifier. In addition, shaders must return a `float4` or a `clamp4` type:

```
float func () { return ...; }                    // an ordinary function
surface shader float4 surf () { return ...; }    // a surface shader
light shader float4 light () { return ...; }     // a light shader
```

The `surface` and `light` modifiers may also be applied to functions. When this is done, such a function may access special features (variables and such) available only to surface and light shaders. In addition, the function becomes accessible only to other surface or light functions and shaders, as appropriate. More examples:

```
surface float surffunc () { return ...; }    // a surface function
surface float lightfunc () { return ...; }   // a light function
```

To query light sources, surface shaders (and functions) use the `integrate()` operator. This operator takes an expression and loops over all active light sources, evaluating the expression once per light source. The operator returns the sum of the expression evaluations.

The `integrate()` operator evaluates special "per-light" expressions, which are expressions that depend directly on special built-in per-light values (in particular the light vector, the half-angle vector, and the light intensity) and/or other per-light expressions. In evaluating a per-light expression once per light, the `integrate()` operator removes the per-light attribute of the integrated expression.

We use a type modifier scheme to track per-light expressions. Just as every value in our system has a type, every value also has a type modifier that specifies whether or not the value changes with every light. In our system, the keyword `perlight` is used to indicate such a value. We require all variables and return values that hold per-light values to be declared with the perlight modifier. We impose this requirement to make user code more readable. Our compiler separately infers which values are `perlight`, and it uses this information to report an error when a `perlight` value is stored to a non-`perlight` variable.

Here are some examples of `perlight` values and the `integrate()` operator. Assume `L`, `H`, and `Cl` are per-light values:

```
float4 Kd = ...;                            // compute diffuse surface color
perlight float NdotL = max(dot(N,L),0);     // max(dot(N,L),0) is perlight
perlight float intensity = Cl * NdotL;      // Cl * NdotL is perlight
float color = Kd * integrate(intensity);    // integrate light and modulate

perlight float NdotH = dot(N,H);    // dot(N,H) is perlight
float NdotH = dot(N,H);             // error: missing perlight modifier
```

As we will see in a later section on built-in global values, `Cl` in particular references the amount of light incident on the surface from each light. By referencing `Cl`, surface shaders indirectly reference the active light shaders.

Values that have been integrated once cannot be integrated again. This is something of an artificial restriction that was imposed, because it really doesn't make a lot of sense to integrate a value that has already been integrated.

4 Computation Frequencies

A key aspect of our system is its support for computations at a variety of different rates, or computation frequencies. We support four different computation frequencies: once at compile time, once per group of primitives, once per vertex, and once per fragment. In our system every shading computation occurs at one of the rates.

Note that we do not provide a frequency that corresponds to once per primitive. Ideally we would support such a frequency, in particular for flat shading, but do not because OpenGL only provides limited support for that computation frequency. Specifically, OpenGL does not provide support for per-primitive texture coordinates.

4.1 Frequency type modifiers

As with our treatment of per-light expressions, we use a type modifier system to control the frequencies at which computations occur. This modifier specifies how often that value is computed (or specified, if the value is a parameter).

There is one type modifier for each computation frequency. The modifiers are: `constant`, `vertex`, `primitive group`, and `fragment`. We provide an additional modifier, `perbegin`, for compatibility with the previous language version. This additional modifier is equivalent to the `primitive group` modifier.

Three base types, namely the two matrix types and the `texref` type, have a maximum computation frequency of primitive group. This restriction effectively limits how often matrices and `texref`s may be computed or specified. This is somewhat of an arbitrary restriction for the matrix types, since there is no reason matrices cannot be computed per-vertex or per-fragment; however, we impose this restriction to simplify our compiler somewhat. The restriction on `texref`s reflects the fact that in OpenGL, textures are specified for entire primitive groups and never more often (such as per-vertex).

Our language defines a set of rules to allow compilers to infer how often a particular value is computed. Such a set of rules is important both because it removes the need for the user to explicitly manage computation frequencies and because it allows for efficient generation of code when the user does not know the computation frequencies of certain values, in particular the intensity of light arriving at a surface, which can reasonably have any computation frequency. In the latter case, a compiler that can infer computation frequencies can properly choose, for example, vertex operations or fragment operations to integrate vertex and fragment lights, respectively.

4.2 Computation frequency inferrence rules

Two rules are used to infer computation frequencies. The first deals with the default computation frequencies of shader parameters, while the second deals with the propagation of computation frequencies across operators. By applying these rules, a compiler can always infer the computation frequency of a given operation.

All shader parameters have a well-defined default computation frequency that indicates how often the parameter may be specified. This frequency depends on the parameter's base type and the corresponding shader's type (surface or light):

Type	Default for surfaces	Default for lights
`bool`	vertex	primitive group
`clampf1`	vertex	primitive group
`clampf3`	vertex	primitive group
`clampf4`	vertex	primitive group
`float1`	vertex	primitive group
`float3`	vertex	primitive group
`float4`	vertex	primitive group
`matrix3`	primitive group	primitive group
`matrix4`	primitive group	primitive group
`texref`	primitive group	primitive group

Note that the defaults are different for surfaces and lights. This reflects the fact that typically light properties do not change more often than per-primitive-group.

The default shader parameter computation frequencies take effect when no computation frequency is specified with the parameter. An explicitly-specified computation frequency overrides the default.

Some examples:

```
surface shader float4 surf1 (float1 f) { ... }      // f is vertex
surface shader float4 surf2 (matrix3 m) { ... }     // m is primitive group
light shader float light1 (float1 f) { ... }        // f is primitive group
light shader float light2 (vertex float1 f) { ... } // f is vertex
light shader float light3 (matrix3 m) { ... }       // m is primitive group
```

Note that the rules for default computation frequencies do not apply to functions. They only apply to shaders:

```
surface surffunc1 (float1 f) { ... }   // no default computation frequency
```

In this case, the computation frequency of `f` is determined by the value passed to `f` when `surffunc1` is called.

The computation frequencies of computed values are determined by applying a second rule that propagates computation frequencies across operators. For the most part, we try to compute things as infrequently as possible. Specifically, the computation frequency of a computed value is the least frequent computation frequency possible given the constraint that a value must be computed at least as often as the most frequent value it depends on. For example, the result of adding a vertex value to another vertex value is a vertex value, but adding a vertex value to a fragment value results in a fragment value, both because of the rule previously mentioned, and because really it doesn't make any sense to try to obtain vertex values from fragment ones.

A number of operations can only be evaluated at certain computation frequencies. For example, texturing can only be computed per-fragment, while matrix-matrix multiplication can be computed at most per-primitive-group. We place additional constraints on computation frequencies to satisfy the limitations of each operation. We describe the details of these per-operator constraints in the operator appendix, Section 9.1.

4.3 Explicitly specifying computation frequencies

While the computation frequencies of computed values are inferred using the rules just described, they may be controlled by explicitly specifying computation frequencies. For example, if two vertex values `N` and `L` are to be used to compute `dot(N,L)`, the result of the dot product will normally be per-vertex. However, a per-fragment dot product can be achieved by first casting `N` or `L` (or both) to a fragment value:

```
float3 Nf = (fragment float3) N;   // cast N, fragment Nf inferred
float3 Lf = (fragment float3) L;   // cast L, fragment Lf inferred
// compute and use dot(Nf,Lf)...

fragment float3 Nf = N;            // use implicit cast from assign
fragment float3 Lf = L;            // use implicit cast from assign
// compute and use dot(Nf,Lf)...

dot(N, (fragment float3)L)...      // cast L only
```

In all three cases, once a fragment version of `N` or `L` is computed, the resulting dot product is inferred to be evaluated per-fragment.

5 Type conversion

A number of type conversions are permitted, including conversion of clamped values to float values, conversion of float values to clamped values, conversion from one computation frequency to a more-frequency computation frequency, and conversion of non-per-light values to per-light values.

Converting clamped values to float values has no effect except perhaps one of number representation (specifically, floating point or fixed point). Also, since floating-point values are more general than clamped floating-point values, this conversion is considered a promotion. Before performing an operation that involves both clamped and unclamped values, clamped values are automatically promoted to unclamped values.

Converting a float value to a clamped value clamps the float value to $[0, 1]$. The number representation possibly changes also. This conversion may be performed explicitly using a type cast, or implicitly when assigning a `float` value to a `clampf` variable.

Conversion from one computation frequency to another is only possible if the new computation frequency is more frequent than the old one. In most cases, such a conversion simply replicates the old value at the new computation frequency; however, the conversion from vertex to fragment is special. In this case, vertex values are interpolated between vertices to obtain a fragment value. The exact nature of the interpolation is currently being left unspecified. Our compiler follows what OpenGL specifies, i.e. texture coordinates are perspective-correct while color values are not necessarily that way.

The conversion of the computation frequencies of operands to an operator is performed automatically as necessary for each operator. This process follows the rules for operator overloading and the function prototypes for operators discussed in later sections.

A non-per-light value may be converted into a per-light value. Performing this conversion has the effect of replicating the non-per-light value for every light.

Unlike in C, there is no way to interpret the value of a comparison numerically.

6 Global variables

Our system supports user-defined global variables as long as they are constant and their values are specified. Globals must be explicitly declared as constant:

```
constant float4 Red = { 1, 0, 0, 1 };  // valid
constant float4 Red;                   // error: missing definition
float4 Red = { 1, 0, 0, 1 };           // error: missing constant keyword

constant float4 DarkRed = 0.5 * Red;   // functions of constants are valid
```

A number of global values are predefined and initialized on demand before a shader executes, or, in the case of predefined `perlight` globals, before each evaluation of the expression integrated by the corresponding `integrate()` operator. The predefined light shader global variables are:

```
vertex float3 S;           // light-space surface vector, normalized
vertex float Sdist;        // distance to surface point
```

The predefined surface shader globals are:

```
vertex float3 N;           // eye-space normal vector, normalized
vertex float3 T;           // eye-space tangent vector, normalized
vertex float3 B;           // eye-space binormal vector, normalized
vertex float3 E;           // eye-space eye vector, normalized

vertex float4 P;           // eye-space surface position, w=1
vertex float4 Pobj;        // object-space surface position, w=1

perbegin float4 Ca;        // color of global ambient light

vertex float4 Cprev;       // previous framebuffer color

vertex perlight float3 L;  // eye-space light vector, normalized
vertex perlight float3 H;  // eye-space halfangle vector, normalized

vertex perlight float4 Cl; // color of light (from a light shader)
```

Note that the definitions of the various globals currently cause light shaders to be evaluated in light space and surface shaders to be evaluated in eye space. Light space is defined by the light's position and orientation, while eye space is defined by the viewer's position and orientation.

The use of built-in parameters implicitly makes a shader dependent on one or more implicit shader parameters which are used to evaluate the built-in parameters. It is important to recognize these implicit shader parameters even though they are not a formal part of the language, since ultimately the user must set these parameters in addition to all those explicitly required by the active surface and light shaders. The implicit parameters are:

```
perbegin float4 __ambient;            // color of global ambient light
perbegin matrix4 __modelview;         // modelview matrix
perbegin matrix4 __projection;        // projection matrix

vertex float3 __normal;               // object-space normal vector
vertex float3 __tangent;              // object-space tangent vector
vertex float3 __binormal;             // object-space binormal vector
vertex float4 __position;             // object-space surface position

perbegin perlight float4 __lightpos;  // homogeneous position of light
perbegin perlight float3 __lightdir;  // unnormalized eye-space light direction
perbegin perlight float3 __lightup;   // unnormalized eye-space light up vector
```

Perlight built-in parameters must be specified once per active light shader.

Note that all shaders depend on `__modelview`, `__projection`, and `__position`.

7 Function Overloading

Our language allows functions to be overloaded in a manner similar to C++. Overloading allows for many functions to be available when a function is called. Availability is defined as a function with the same name and number of parameters. We define a set of rules to select which function to select when more than one choice is available. The rules examine the base types of the parameters used in the call to form groups of matching functions.

The first group consists of functions whose parameter base types match the base types of the parameters in the call exactly.

The second group consists of functions whose parameter base types match the base types of the parameters in the call through the possible use of promotion. In particular, we consider the promotion of clamped floats to floats to form matches.

The third group consists of functions whose parameter base types match the base types of the parameters in the call through the use of both promotion and demotion.

The first group is checked first. If empty, the second group is checked, and likewise for the third group. If all three groups are empty, there is no match, and an error is generated. If any group being checked has more than one choice available, the call is ambiguous, and an error is generated. A match is found only if exactly one match is available in the first non-empty group.

This overloading mechanism is used for user-defined functions as well as built-in functions and built-in operators. Built-in functions and operators are defined using function prototypes in the operator appendix, Section 9.1.

8 Conditional Compilation

Today's hardware platforms offer differing sets of functionality. Some operators are not available on all hardware. To solve this problem, our language supports conditional compilation using a very-limited subset of C-preprocessor directives. We support:

```
#if <integer>
#ifdef <identifier>
#ifndef <identifier>
#else
#endif
#define <identifier>
#undef <identifier>
```

To promote the creation of function libraries, we also provide a limited include directive:

```
#include "<filename>"
```

We only support relative filenames, which must be double-quoted. We do not support angle-bracked filenames for searching include directories.

Our compiler predefines a number of identifiers based on whether or not certain hardware features are available. These identifiers are:

- `HAVE_FRAGMENT_SUBTRACT`. Indicates whether or not the subtract operator is available per-fragment.
- `HAVE_TEXTURE_3D`. Indicates whether or not the `texture3d` operator is available.
- `HAVE_CUBEMAP`. Indicates whether or not the `cubemap` operator is avialble.
- `HAVE_BUMPOPS`. Indicates whether or not the `bumpdiff` and `bumpspec` operators are available.
- `HAVE_REGISTER_COMBINERS`. Covers the availability of the following operators per-fragment: `dot`, `select`, `rgb`, `blue`, `alpha`, `lthalf`, `cubenorm`.
- `HAVE_FRAGMENT_INDEX`. Indicates whether or not the `[]` operator is available per-fragment.
- `HAVE_FRAGMENT_COMPARES`. Indicates whether or not the `==`, `!=`, `>`, `<`, `>=`, and `<=` operators are available per-fragment.
- `HAVE_FRAGMENT_PROGRAM`. Covers availability of the following operators per-fragment: `dot`, `select`, `rgb`, `blue`, `alpha`, `lthalf`, arithmetic operators (add, subtract, multiply, divide), `join`, `swizzle`, `normalize`, `cross`, `length`, `max`, `min`, `pow`, `reflect`, `sqrt`, `sin`, `cos`, `ceil`, `floor`, `trunc`, `mod`, `dx`, `dy`, `xyz_screen`, and assignment mask.

9 Appendices

9.1 Built-in operators and functions

In this appendix, we describe the enumerate the built-in operators and functions made available by our language. Except for the syntax by which they are referred to, built-in operators and functions behave identically.

Every built-in operator and function has a range of computation frequencies at which it may be evaluated; the range specifies both a minimum and a maximum frequency.

As described earlier, values are evaluated as infrequently as possible. We define this computation frequency precisely as the maximum frequency among all of an operator's operands and the operator's miminum computation frequency.

Minimum and maximum computation frequencies limit the kinds of operations available at each computation frequency. For example, they restrict many matrix manipulation operations to a maximum computation frequency of per-primitive-group, and they force texture mapping to be per-fragment.

An error is generated if an operator's evaluation computation frequency exceeds the operator's maximum computation frequency.

In addition to each operator having a range of computation frequencies, every operand of every operator also has an associated range of computation frequencies. In most cases, this range has a minimum frequency of constant and a maximum frequency equal to the maximum frequency of the operator itself, but in a few cases, the range is more restrictive. For example, current hardware does not support the use of per-fragment texture coordinates. We therefore limit the maximum computation frequency of texture coordinates to vertex values.

In cases where the minimum frequency of an operand is not met, the value passed to the operand is automatically cast to an appropriate computation frequency. In cases where the maximum frequency of an operand is exceeded, an error is generated.

Not all operations are supported by all hardware at all computation frequencies. The compiler is allowed to generate an error when an unsupported operation is used. The section regarding conditional compilation enumerates the most important sets of operators that fall into this category.

We now list all of the available operators. In the listings below, ranges are specified using a [min:]max syntax. For operators, if the min is unspecified, it defaults to `constant`. For operands, if the min and max are unspecified, the range defaults to the range of the corresponding operator, otherwise if only the min is unspecified, the min defaults to the max.

```
fragment float1 operator+ (float1, float1)
fragment float3 operator+ (float3, float3)
fragment float4 operator+ (float4, float4)
fragment clampf1 operator+ (clampf1, clampf1)
fragment clampf3 operator+ (clampf3, clampf3)
fragment clampf4 operator+ (clampf4, clampf4)

fragment float1 operator- (float1, float1)
fragment float3 operator- (float3, float3)
fragment float4 operator- (float4, float4)
fragment clampf1 operator- (clampf1, clampf1)
fragment clampf3 operator- (clampf3, clampf3)
fragment clampf4 operator- (clampf4, clampf4)

fragment float1 operator* (float1, float1)
fragment float3 operator* (float3, float3)
fragment float3 operator* (float1, float3)
fragment float3 operator* (float3, float1)
fragment float4 operator* (float4, float4)
fragment float4 operator* (float1, float4)
fragment float4 operator* (float4, float1)
fragment clampf1 operator* (clampf1, clampf1)
fragment clampf3 operator* (clampf3, clampf3)
fragment clampf3 operator* (clampf1, clampf3)
fragment clampf3 operator* (clampf3, clampf1)
fragment clampf4 operator* (clampf4, clampf4)
fragment clampf4 operator* (clampf1, clampf4)
fragment clampf4 operator* (clampf4, clampf1)
perbegin matrix3 operator* (matrix3, matrix3)
perbegin matrix4 operator* (matrix4, matrix4)
vertex float3 operator* (matrix3, float3)
```

```
vertex float4 operator* (matrix4, float4)

fragment float1 operator/ (float1, float1)
fragment float3 operator/ (float3, float3)
fragment float3 operator/ (float1, float3)
fragment float3 operator/ (float3, float1)
fragment float4 operator/ (float4, float4)
fragment float4 operator/ (float1, float4)
fragment float4 operator/ (float4, float1)
fragment clampf1 operator/ (clampf1, clampf1)
fragment clampf3 operator/ (clampf3, clampf3)
fragment clampf3 operator/ (clampf1, clampf3)
fragment clampf3 operator/ (clampf3, clampf1)
fragment clampf4 operator/ (clampf4, clampf4)
fragment clampf4 operator/ (clampf1, clampf4)
fragment clampf4 operator/ (clampf4, clampf1)

fragment float1 operator/ (float1, float1)
fragment float3 operator/ (float3, float3)
fragment float3 operator/ (float1, float3)
fragment float3 operator/ (float3, float1)
fragment float4 operator/ (float4, float4)
fragment float4 operator/ (float1, float4)
fragment float4 operator/ (float4, float1)
fragment clampf1 operator/ (clampf1, clampf1)
fragment clampf3 operator/ (clampf3, clampf3)
fragment clampf3 operator/ (clampf1, clampf3)
fragment clampf3 operator/ (clampf3, clampf1)
fragment clampf4 operator/ (clampf4, clampf4)
fragment clampf4 operator/ (clampf1, clampf4)
fragment clampf4 operator/ (clampf4, clampf1)

fragment float1 operator- (float1)
fragment float3 operator- (float3)
fragment float4 operator- (float4)

fragment float1 operator[] (float3)
fragment float1 operator[] (float4)
fragment clampf1 operator[] (clampf3)
fragment clampf1 operator[] (clampf4)
perbegin float3 operator[] (matrix3)
perbegin float4 operator[] (matrix4)

fragment float3 operator[] (float1)
fragment float3 operator[] (float3)
fragment float3 operator[] (float4)
fragment clampf3 operator[] (clampf1)
fragment clampf3 operator[] (clampf3)
fragment clampf3 operator[] (clampf4)
fragment float4 operator[] (float1)
fragment float4 operator[] (float3)
fragment float4 operator[] (float4)
fragment clampf4 operator[] (clampf1)
fragment clampf4 operator[] (clampf3)
fragment clampf4 operator[] (clampf4)

fragment float3 operator writemask (float3)
fragment float4 operator writemask (float4)
```

```
vertex float3 operator{} (float, float, float)
vertex float4 operator{} (float, float, float, float)
vertex clampf3 operator{} (clampf, clampf, clampf)
vertex clampf4 operator{} (clampf, clampf, clampf, clampf)

fragment float3 operator{} (float, float, float)
fragment float4 operator{} (float, float, float, float)
fragment clampf3 operator{} (clampf, clampf, clampf)
fragment clampf4 operator{} (clampf, clampf, clampf, clampf)

fragment float4 operator{} (float3 rgb, float1 alpha)
fragment clampf4 operator{} (clampf3 rgb, clampf1 alpha)
perbegin matrix3 operator{} (float3, float3, float3)
perbegin matrix4 operator{} (float4, float4, float4, float4)

fragment bool operator== (float, float)
fragment bool operator!= (float, float)
fragment bool operator> (float, float)
fragment bool operator< (float, float)
fragment bool operator>= (float, float)
fragment bool operator<= (float, float)

fragment bool operator== (clampf, clampf)
fragment bool operator!= (clampf, clampf)
fragment bool operator> (clampf, clampf)
fragment bool operator< (clampf, clampf)
fragment bool operator>= (clampf, clampf)
fragment bool operator<= (clampf, clampf)

fragment bool operator and (bool, bool)
fragment bool operator or (bool, bool)
fragment bool operator xor (bool, bool)
fragment bool operator not (bool, bool)

fragment float4 operator blend (float4, float4)
fragment clampf4 operator blend (clampf4, clampf4)
fragment float4 operator over (float4, float4)
fragment clampf4 operator over (clampf4, clampf4)
fragment float4 operator blend_over (float4, float4)
fragment clampf4 operator blend_over (clampf4, clampf4)

surface fragment float1 operator integrate (float1)
surface fragment float3 operator integrate (float3)
surface fragment float4 operator integrate (float4)
surface fragment clampf1 operator integrate (clampf1)
surface fragment clampf3 operator integrate (clampf3)
surface fragment clampf4 operator integrate (clampf4)

fragment bool operator () (bool)
fragment float operator () (float)
fragment float3 operator () (float3)
fragment float4 operator () (float4)
fragment clampf operator () (clampf)
fragment clampf3 operator () (clampf3)
fragment clampf4 operator () (clampf4)
perbegin matrix3 operator () (matrix4)
perbegin matrix4 operator () (matrix4)
perbegin texref operator () (texref)
```

```
constant matrix3 identity3 ()
constant matrix4 identity ()

perbegin matrix3 affine (matrix4)
perbegin matrix3 invert (matrix3)
perbegin matrix3 rotate3 (float angle, float x, float y, float z)
perbegin matrix3 scale3 (float x, float y, float z)
perbegin matrix3 transpose (matrix3)
perbegin matrix4 frustum (float l, float r, float b, float t, float n, float f)
perbegin matrix4 invert (matrix4)
perbegin matrix4 lookat (float ex, float ey, float ez, float cx, float cy,
                         float cz, float ux, float uy, float uz)
perbegin matrix4 ortho (float l, float r, float b, float t, float n, float f)
perbegin matrix4 rotate (float angle, float x, float y, float z)
perbegin matrix4 scale (float x, float y, float z)
perbegin matrix4 translate (float x, float y, float z)
perbegin matrix4 transpose (matrix4)

fragment float clamp (float val, float lo, float hi)
fragment float3 clamp (float3 val, float lo, float hi)
fragment float3 clamp (float3 val, float3 lo, float3 hi)
fragment float4 clamp (float4 val, float lo, float hi)
fragment float4 clamp (float4 val, float4 lo, float4 hi)
fragment float3 cross (float3, float3)
fragment float dot (float3, float3)
fragment float dot (float4, float4)
fragment float length (float3)
fragment float length (float4)
fragment float max (float, float)
fragment float3 max (float3, float3)
fragment float4 max (float4, float4)
fragment float min (float, float)
fragment float3 min (float3, float3)
fragment float4 min (float4, float4)
fragment float3 normalize (float3)
fragment float4 normalize (float4)
fragment float pow (float val, float exp)
fragment float3 reflect (float3 vec, float3 norm)
fragment float sqrt (float)
fragment float cos (float)
fragment float sin (float)
fragment float ceil (float)
fragment float floor (float)
fragment float mod (float, float)
fragment float trunc (float)

fragment float dx (float)
fragment float3 dx (float3)
fragment float4 dx (float4)
fragment float dy (float)
fragment float3 dy (float3)
fragment float4 dy (float4)

fragment float4 xyz_screen ()

fragment float1 select (bool, float1, float1)
fragment float3 select (bool, float3, float3)
fragment float4 select (bool, float4, float4)
fragment clampf1 select (bool, clampf1, clampf1)
```

```
fragment clampf3 select (bool, clampf3, clampf3)
fragment clampf4 select (bool, clampf4, clampf4)
fragment float3 rgb (float1)
fragment float3 rgb (float4)
fragment clampf3 rgb (clampf1)
fragment clampf3 rgb (clampf4)
fragment float1 blue (float3)
fragment float1 blue (float4)
fragment clampf1 blue (clampf3)
fragment clampf1 blue (clampf4)
fragment float1 alpha (float4)
fragment clampf1 alpha (clampf4)
fragment bool lthalf (float1)
fragment bool lthalf (clampf1)

fragment:fragment clampf4 lut (fragment clampf4)
fragment:fragment clampf4 texture (texref tex, constant:vertex float3 coord)
fragment:fragment clampf4 texture (texref tex, constant:vertex float4 coord)
fragment:fragment clampf4 texture3d (texref tex, constant:vertex float3 coord)
fragment:fragment clampf4 texture3d (texref tex, constant:vertex float4 coord)
fragment:fragment clampf4 cubemap (texref ref, constant:vertex float3 coord)
fragment:fragment clampf4 cubemap (texref ref, constant:vertex float4 coord)
fragment:fragment clampf3 cubenorm (constant:vertex float3 vec)
fragment:fragment clampf4 bumpdiff (texref ref, constant:vertex float4 coord,
                                    constant:vertex float3 Ltan)
fragment:fragment clampf4 bumpspec (texref ref, constant:vertex float4 coord,
                                    constant:vertex float3 Htan)
```

9.2 Grammar

The following grammar describes the overall organization of the language.

```
PROGRAM : DECL_LIST

DECL_LIST : DECL_LIST DECL

DECL : TYPE IDENT ;
     | TYPE IDENT = EXPR ;
     | TYPE IDENT ( PARAM_LIST ) { STMT_LIST }

TYPE : MOD_LIST BASE_TYPE

MOD_LIST : MOD_LIST MOD

MOD : constant | primitive group | vertex | fragment | light | surface |
      shader | perlight | perbegin

BASE_TYPE : bool | clampf | clampf1 | clampf3 | clampf4 | clampfv |
            float | float1 | float3 | float4 | floatv | matrix3 | matrix4 |
            matrix | texref

PARAM_LIST : PARAM
           | PARAM_LIST ',' PARAM

PARAM : TYPE IDENT

STMT_LIST : STMT_LIST STMT

STMT : TYPE IDENT ;
```

```
     | TYPE IDENT = EXPR ;
     | EXPR ;
     | return EXPR ;
     | ;

EXPR : UNARY = EXPR
     | EXPR BINOP EXPR
     | UNARY

BINOP : == | != | > | < | >= | <= | + | - | blend | over | blend_over | * | /

UNARY : - UNARY
      | ( TYPE ) UNARY
      | PRIMARY

PRIMARY : ( EXPR )
        | { EXPR_LIST }
        | IDENT
        | PRIMARY [ INTEGER ]
        | PRIMARY [ INTEGER, INTEGER, INTEGER ]
        | PRIMARY [ INTEGER, INTEGER, INTEGER, INTEGER ]
        | integrate ( EXPR )
        | IDENT ( EXPR_LIST )
        | INTEGER
        | FLOAT

EXPR_LIST : EXPR
          | EXPR_LIST , EXPR
```

The following non-terminals are described by regular expressions:

```
IDENT : [_a-zA-Z][_a-zA-Z0-9]*
INTEGER : [0-9]+
FLOAT : (([0-9]+(\.[0-9]*)?)|(\.[0-9]+))([eE][-+]?[0-9]+)?f?
```

9.3 Sample shaders

The following example shaders serve to illustrate how the shading language might be used to implement a number of interesting shading effects.

```
// Useful constants

constant float4 Zero = { 0, 0, 0, 0 };
constant float4 Black = { 0, 0, 0, 1 };
constant float4 White = { 1, 1, 1, 1 };

constant float pi = 3.14159;

// Light shaders

light float
atten (float ac, float al, float aq)
{
    return 1.0 / ((aq * Sdist + al) * Sdist + ac);
}

light shader float4
simple_light (float4 color, float ac, float al, float aq)
{
```

```
    return color * atten(ac, al, aq);
}

float
smoothstep (float value, float min, float max)
{
    float t = clamp((value - min) / (max - min), 0, 1);
    return t * t * (3 - 2 * t);
}

float
smoothspot (float spot_cos, float inner_edge_angle, float outer_edge_angle)
{
    float inner_cos = cos(inner_edge_angle * pi / 180);
    float outer_cos = cos(outer_edge_angle * pi / 180);
    return smoothstep(spot_cos, outer_cos, inner_cos);
}

light shader float4
spotlight (float4 color, float ac, float al, float aq)
{
    float4 Cl = smoothspot(-S[2], 15, 30) * color * atten(ac, al, aq);
    return Cl;
}

light float4
star_projector_f (float4 color, float ac, float al, float aq, texref stars,
                  float time)
{
    float4 Cl = smoothspot(-S[2], 15, 30) * color * atten(ac, al, aq);
    float4 uv = { S[0], S[1], 0, -S[2] }; // project
    matrix4 t_rot = rotate(time * 15, 0, 0, 1);
    return Cl * texture(stars, t_rot * scale(1.5, 1.5, 1) * uv);
}

light shader float4
star_projector (float4 color, float ac, float al, float aq, texref stars)
{
    return star_projector_f(color, ac, al, aq, stars, 0);
}

light shader float4
star_projector_anim (float4 color, float ac, float al, float aq, texref stars,
                     float time)
{
    return star_projector_f(color, ac, al, aq, stars, time);
}

// Reflection models

surface float4
lightmodel (float4 a, float4 d, float4 s, float4 e, float sh)
{
    perlight float diffuse = dot(N,L);
    perlight float specular = pow(max(dot(N,H),0),sh);
    perlight float4 fr = select(diffuse > 0, d * diffuse + s * specular, Zero);
    return a * Ca + integrate(fr * Cl) + e;
}
```

```
surface float4
lightmodel_diffuse (float4 a, float4 d)
{
    perlight float diffuse = dot(N,L);
    perlight float4 fr = select(diffuse > 0, d * diffuse, Zero);
    return a * Ca + integrate(fr * Cl);
}

surface float4
lightmodel_specular (float4 s, float4 e, float sh)
{
    perlight float diffuse = dot(N,L);
    perlight float specular = pow(max(dot(N,H),0),sh);
    perlight float4 fr = select(diffuse > 0, s * specular, Zero);
    return integrate(fr * Cl) + e;
}

surface float4
lightmodel_anisotropic_u (float4 a, float4 d, float4 s, float4 e, float sh)
{
    float EdotT = dot(E,T);
    perlight float LdotT = dot(L,T);
    perlight float diff = sqrt(1 - LdotT * LdotT);
    perlight float spec = max(diff * sqrt(1 - EdotT*EdotT) - LdotT*EdotT, 0);
    perlight float4 fr = max(dot(N,L),0) * (d * diff + s * pow(spec,sh));
    return a * Ca + integrate(fr * Cl) + e;
}

surface float4
lightmodel_anisotropic_v (float4 a, float4 d, float4 s, float4 e, float sh)
{
    float EdotB = dot(E,B);
    perlight float LdotB = dot(L,B);
    perlight float diff = sqrt(1 - LdotB*LdotB);
    perlight float spec = max(diff * sqrt(1 - EdotB*EdotB) - LdotB*EdotB, 0);
    perlight float4 fr = max(dot(N,L),0) * (d * diff + s * pow(spec,sh));
    return a * Ca + integrate(fr * Cl) + e;
}

float center (float value) { return 0.5 * value + 0.5; }

surface float4
lightmodel_textured_anisotropic_u (texref anisotex, float4 a, float4 e)
{
    perlight float4 uv = { center(dot(T,E)), center(dot(T,L)), 0, 1 };
    // moving Cl helps group vertex/fragment computations
    //perlight float4 fr = max(dot(N,L),0) * texture(anisotex, uv);
    //return a * Ca + integrate(Cl * fr) + e;
    perlight float4 clfr = Cl * max(dot(N,L),0) * texture(anisotex, uv);
    return a * Ca + integrate(clfr) + e;
}

surface float4
lightmodel_textured_anisotropic_v (texref anisotex, float4 a, float4 e)
{
    perlight float4 uv = { center(dot(B,E)), center(dot(B,L)), 0, 1 };
    // moving Cl helps group vertex/fragment computations
    //perlight float4 fr = max(dot(N,L),0) * texture(anisotex, uv);
    //return a * Ca + integrate(Cl * fr) + e;
```

```
    perlight float4 clfr = Cl * max(dot(N,L),0) * texture(anisotex, uv);
    return a * Ca + integrate(clfr) + e;
}

surface float4
lightmodel_cartoon (texref cartoon, float4 a, float4 d)
{
    perlight float fr = max(dot(N,L),0);
    // clamp upper end to avoid texture border color
    float4 uv = { min(integrate(fr) + 0.2, 0.75), 0, 0, 1 };
    return a * Ca + d * texture(cartoon, uv);
}

// Standard material properties

constant float4 Ma = { 0.35, 0.35, 0.35, 1.00 };
constant float4 Md = { 0.50, 0.50, 0.50, 1.00 };
constant float4 Ms = { 1.00, 1.00, 1.00, 1.00 };
constant float4 Me = { 0.00, 0.00, 0.00, 0.00 };
constant float Msh = 300;

surface shader float4
default ()
{
    return lightmodel(Ma, Md, Ms, Me, Msh);
}

surface shader float4
cartoontest (texref cartoon)
{
    return lightmodel_cartoon(cartoon, {.4, .4, .8, 1}, {.4, .4, .8, 1});
}

surface shader float4
bowling_pin (texref pinbase, texref bruns, texref circle, texref coated,
             texref marks, float4 uv)
{
    float4 uv_wrap = { uv[0], 10 * Pobj[1], 0, 1 };
    float4 uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
    matrix4 t_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
    matrix4 t_bruns = invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
    matrix4 t_circle = invert(translate(-0.8, -1.15, 0) * scale(1.4, 1.4, 1));
    matrix4 t_coated = invert(translate(2.6, -2.8, 0) * scale(-5.2, 5.2, 1));
    matrix4 t_marks = invert(translate(2.0, 7.5, 0) * scale (4, -15, 1));
    float front = select(Pobj[2] >= 0, 1, 0);
    float back = select(Pobj[2] <= 0, 1, 0);
    float4 Base = texture(pinbase, t_base * uv_wrap);
    float4 Bruns = front * texture(bruns, t_bruns * uv_label);
    float4 Circle = front * texture(circle, t_circle * uv_label);
    float4 Coated = back * texture(coated, t_coated * uv_label);
    float4 Marks = texture(marks, t_marks * uv_wrap);
    float4 Cd = lightmodel_diffuse({ 0.4, 0.4, 0.4, 1 }, { 0.5, 0.5, 0.5, 1 });
    float4 Cs = lightmodel_specular({ 0.35, 0.35, 0.35, 1 }, Zero, 20);
    return (Circle over (Bruns over (Coated over Base))) * (Marks * Cd) + Cs;
}

surface shader float4
glossy_moons (texref gloss, float4 uv)
{
```

```
    float4 base_a = { 0.1, 0.1, 0.1, 1.00 };
    float4 base_d = { 0.70, 0.40, 0.10, 1.00 };
    float4 base_s = { 0.07, 0.04, 0.01, 1.00 };
    float4 base_e = { 0.00, 0.00, 0.00, 1.00 };
    float base_sh = 15;

    float4 gloss_a = { 0.07, 0.04, 0.01, 1.00 };
    float4 gloss_d = { 0.07, 0.04, 0.01, 1.00 };
    float4 gloss_s = { 1.00, 0.90, 0.60, 1.00 };
    float4 gloss_e = { 0.00, 0.00, 0.00, 1.00 };
    float gloss_sh = 25;

    float4 Cbase = lightmodel(base_a, base_d, base_s, base_e, base_sh);
    float4 Cgloss = lightmodel(gloss_a, gloss_d, gloss_s, gloss_e, gloss_sh);

    float4 uv_gloss = invert(scale(.335,.335,1)) * uv;
    return Cbase + Cgloss * texture(gloss, uv_gloss);
}

surface shader float4
anisotropic_ball_vertex (texref star)
{
    float4 Ma = { 0.1, 0.1, 0.1, 1.0 };
    float4 Md = { 0.3, 0.3, 0.3, 1.0 };
    float4 Ms = { 0.7, 0.7, 0.7, 1.0 };
    float4 Me = { 0.0, 0.0, 0.0, 0.0 };
    float Msh = 15;
    float4 base = texture(star, { center(Pobj[2]), center(Pobj[0]), 0, 1 });
    return base * lightmodel_anisotropic_v(Ma, Md, Ms, Me, Msh);
}

surface shader float4
anisotropic_ball_texture (texref star, texref anisotex)
{
    float4 Ma = { 0.1, 0.1, 0.1, 1.0 };
    float4 Me = { 0.0, 0.0, 0.0, 0.0 };
    float4 base = texture(star, { center(Pobj[2]), center(Pobj[0]), 0, 1 });
    return base * lightmodel_textured_anisotropic_v(anisotex, Ma, Me);
}

surface float4
spheremap (texref env)
{
    float3 R = normalize(reflect(E,N) + { 0, 0, 1 });
    float4 uv = { center(R[0]), center(R[1]), 0, 1 };

    return texture(env, uv);
}

surface shader float4
sphere_map_env (texref env)
{
    return spheremap(env);
}

surface shader float4
poolball (texref one, float4 uv)
{
    float4 Ma = { 0.35, 0.35, 0.35, 1.00 };
```

```
    float4 Md = { 0.50, 0.50, 0.50, 1.00 };
    float4 Ms = { 1.00, 1.00, 1.00, 1.00 };
    float4 Me = { 0.00, 0.00, 0.00, 1.00 };
    float Msh = 127;
    float4 Cd = lightmodel_diffuse(Ma, Md);
    float4 Cs = lightmodel_specular(Ms, Me, Msh);
    matrix4 tm = invert(translate(0.35, 0.2, 0.0) * scale(0.3, 0.6, 1.0));
    return Cd * texture(one, tm * uv) + Cs;
}

surface shader float4
poolball_with_env (texref one, texref env, float4 uv)
{
    float4 Ma = { 0.35, 0.35, 0.35, 1.00 };
    float4 Md = { 0.50, 0.50, 0.50, 1.00 };
    float4 Ms = { 1.00, 1.00, 1.00, 1.00 };
    float4 Me = { 0.00, 0.00, 0.00, 1.00 };
    float Msh = 127;
    float4 Cd = lightmodel_diffuse(Ma, Md);
    float4 Cs = lightmodel_specular(Ms, Me, Msh);
    matrix4 tm = invert(translate(0.35, 0.2, 0.0) * scale(0.3, 0.6, 1.0));
    return Cd * texture(one, tm * uv) + (Cs + spheremap(env));
}

float4
turb (texref noise, float4 uv)
{
    float4 uv_0 = invert(rotate(30.2, 0, 0, 1) * scale(4, 4, 1)) * uv;
    float4 uv_1 = invert(rotate(-35.5, 0, 0, 1) * scale(2, 2, 1)) * uv;
    float4 uv_2 = invert(rotate(274.1, 0, 0, 1) * scale(1, 1, 1)) * uv;
    float4 N_0 = 0.57 * texture(noise, uv_0);
    float4 N_1 = 0.29 * texture(noise, uv_1);
    float4 N_2 = 0.14 * texture(noise, uv_2);
    return N_0 + N_1 + N_2;
}

surface shader float4
noise_2d_multipass (texref noise, float4 uv)
{
    return turb(noise, uv);
}

surface shader float4
noise_2d_multipass_specular_modulate (texref noise, float4 uv)
{
    float4 Cl = lightmodel(Ma, Md, Ms, Me, Msh);
    return Cl * turb(noise, uv);
}

surface shader float4
noise_2d_multipass_specular_separate (texref noise, float4 uv)
{
    float4 Cd = lightmodel_diffuse(Ma, Md);
    float4 Cs = lightmodel_specular(Ms, Me, Msh);
    return Cd * turb(noise, uv) + Cs;
}

float4
skymap (texref clouds, float4 dir, float time)
```

```
{
    dir = normalize(dir);
    dir = { dir[0], dir[1], 4 * (dir[2] + 0.707), 0 };
    dir = normalize(dir);
    float4 uv_lo = dir * { 2, 2, 0, 0 } + { time / 15 , time / 15, 0, 1 };
    float4 uv_hi = dir * { 3, 3, 0, 0 } + { time / 15 , time / 15, 0, 1 };
    float4 Lo = texture(clouds, uv_lo);
    float4 Hi = texture(clouds, rotate(125, 0, 0, 1) * uv_hi);
    // for now, do not use Lo over (Hi over { 0.6, 0.5, 1.0, 1.0 })
    // texture_env_combine does not do over correctly
    return Lo over Hi over { 0.6, 0.5, 1.0, 1.0 };
}

surface shader float4
quake_sky (texref clouds, float time)
{
    return skymap(clouds, { Pobj[0], -Pobj[2], Pobj[1], 0 }, time);
}

surface shader float4
bowling_pin_with_sky (texref pinbase, texref bruns, texref circle,
                      texref coated, texref marks, float4 uv,
                      texref clouds, float time)
{
    float4 uv_wrap = { uv[0], 10 * Pobj[1], 0, 1 };
    float4 uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
    matrix4 t_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
    matrix4 t_bruns = invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
    matrix4 t_circle = invert(translate(-0.8, -1.15, 0) * scale(1.4, 1.4, 1));
    matrix4 t_coated = invert(translate(2.6, -2.8, 0) * scale(-5.2, 5.2, 1));
    matrix4 t_marks = invert(translate(2.0, 7.5, 0) * scale (4, -15, 1));
    float front = select(Pobj[2] >= 0, 1, 0);
    float back = select(Pobj[2] <= 0, 1, 0);
    float4 Base = texture(pinbase, t_base * uv_wrap);
    float4 Bruns = front * texture(bruns, t_bruns * uv_label);
    float4 Circle = front * texture(circle, t_circle * uv_label);
    float4 Coated = back * texture(coated, t_coated * uv_label);
    float4 Marks = texture(marks, t_marks * uv_wrap);
    float Lscale = 0.5;
    float4 Cd = lightmodel_diffuse({ 0.4, 0.4, 0.4, 1 }, { 0.5, 0.5, 0.5, 1 });
    Cd = Cd * Lscale;
    float4 Cs = lightmodel_specular({ 0.35, 0.35, 0.35, 1 }, Zero, 20);
    Cs = Cs * Lscale;
    float3 R = reflect(E,N);
    return (Circle over (Bruns over (Coated over Base))) * (Marks * Cd) + Cs +
           0.5 * skymap(clouds, { R[0], -R[2], R[1], 0 }, time);
}

#ifdef HAVE_BUMPOPS

surface shader float4
bowling_pin_bump (texref pinbase, texref bruns, texref circle, texref coated,
                  texref marks, texref marksbump, float4 uv)
{
    float4 uv_wrap = { uv[0], 10 * Pobj[1], 0, 1 };
    float4 uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
    matrix4 t_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
    matrix4 t_bruns = invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
    matrix4 t_circle = invert(translate(-0.8, -1.15, 0) * scale(1.4, 1.4, 1));
```

```
    matrix4 t_coated = invert(translate(2.6, -2.8, 0) * scale(-5.2, 5.2, 1));
    matrix4 t_marks = invert(translate(2.0, 7.5, 0) * scale (4, -15, 1));
    float front = select(Pobj[2] >= 0, 1, 0);
    float back = select(Pobj[2] <= 0, 1, 0);
    float4 Base = texture(pinbase, t_base * uv_wrap);
    float4 Bruns = front * texture(bruns, t_bruns * uv_label);
    float4 Circle = front * texture(circle, t_circle * uv_label);
    float4 Coated = back * texture(coated, t_coated * uv_label);
    float4 uv_marks = t_marks * uv_wrap;
    float4 Marks = texture(marks, uv_marks);
    perlight float3 Lt = { dot(T,L), dot(B,L), dot(N,L) };
    perlight float3 Ht = { dot(T,H), dot(B,H), dot(N,H) };
    float4 Ma = {.4,.4,.4,1};
    float4 Md = {.5,.5,.5,1};
    float4 Ms = {.3,.3,.3,1};
    float4 Kd = (Circle over (Bruns over (Coated over Base))) * Marks;
    return Kd * Ma +
           integrate(Cl * (Kd * Md * bumpdiff(marksbump, uv_marks, Lt)
                           blend(ONE,SRC_ALPHA)
                           Ms * bumpspec(marksbump, uv_marks, Ht)));
}

#endif /* HAVE_BUMPOPS */

#ifdef HAVE_CUBEMAP

surface shader float4
cube_from_obj_normal (texref cube) {
    return cubemap(cube, {-1,-1,1}*__normal);
}

surface shader float4
poolball_with_cube (texref one, float4 uv, texref cube)
{
    float4 Ma = .5 * { 0.35, 0.35, 0.35, 1.00 };
    float4 Md = .5 * { 0.50, 0.50, 0.50, 1.00 };
    float4 Ms = .5 * { 1.00, 1.00, 1.00, 1.00 };
    float4 Me = .5 * { 0.00, 0.00, 0.00, 1.00 };
    float Msh = 127;
    float4 Cd = lightmodel_diffuse(Ma, Md);
    float4 Cs = lightmodel_specular(Ms, Me, Msh);
    matrix4 tm = invert(translate(0.35, 0.2, 0.0) * scale(0.3, 0.6, 1.0));
    float3 R = reflect(E,N);
    return Cd * texture(one, tm * uv) + Cs + 0.4 * cubemap(cube, {-1,-1,1}*R);
}

#endif /* HAVE_CUBEMAP */
```

Shading System Immediate-Mode API v2.2

William R. Mark and C. Philipp Schloter

April 4, 2002

1 Introduction

This document describes modifications to the OpenGL API to support the immediate-mode use of the Stanford real-time shading language. We collectively refer to these extensions as the shading-language immediate-mode API. These extensions are implemented as a layer on top of regular OpenGL.

The immediate-mode API supports the following major operations:

1. Loading the source code for a light shader or surface shader from a file.
2. Associating one or more light shader(s) with a surface shader to create a combined surface/light shader.
3. Compiling a combined surface/light shader for the current graphics hardware.
4. Selecting a compiled surface/light shader to use as the current shader for rendering.
5. Setting values of shader parameters.

When a shader is active, many OpenGL commands are no longer allowed, because their functionality is provided through the shading language. The disallowed commands fall into four major categories:

1. Fragment-processing commands (e.g. fog, texturing modes)
2. Texture-coordinate generation and transformation commands
3. Lighting commands.
4. Material-property commands.

When using our "lburg" multi-pass fragment backend, commands that configure framebuffer blending modes are also forbidden (instead, use the `Cprev` builtin variable within a shader). However, these commands are allowed with our "nv" fragment backend.

Using a forbidden OpenGL command while a programmable-shader is active will result in undefined behavior.

2 Initialization

`sglInit()`

The application must initialize the programmable shading system by calling `sglInit()` once, before calling any other `sgl*` routines.

3 Loading shader source code

`int sglShaderFile(GLuint shaderSourceID, char *shaderName, char *filename)`

Loads the source code for the shader named `shaderName` from file `filename`, and assigns it the identifier `shaderCodeID`. Any other shaders that are specified in the file are ignored. The loaded shader source code becomes the active shader source code. The specified shader may be either a light shader or a surface shader. `shaderCodeID` must be unused when this routine is called. The return code is 0 if there were no errors, 1 if there was an error.

4 Compiling and activating shaders

After the source code for a shader is loaded, but before it is used, the shader must be compiled. Our system treats shader source code and compiled shaders as largely separate entities.

`sglCompileShader(GLuint shaderID)`

Compiles the current shader source code. The compiled shader is assigned the user-specified identifier `shaderID`. If the shader is a surface shader, it incorporates any currently associated light shaders (discussed in the next section).

The newly created shader becomes the 'active' shader, as if sglBindShader() had been called. If the shader is a light shader, it is only active in the sense that subsequent `sglParameterHandle()` calls will apply to it. A light shader can only be activated for rendering purposes by associating it with a surface shader using `sglUseLight()`.

The current shader source code remains unchanged by this call.

Note that `shaderID` may not be -1, because this value is reserved for `SGL_STD_OPENGL`, the standard OpenGL lighting/shading model.

`sglBindShader(GLuint shaderID)`

Changes the currently active shader to that specified by `shaderID`. Note that it is illegal to render geometry when a light shader is bound.

Specifying `SGL_STD_OPENGL` reverts to the standard OpenGL lighting/shading model.

5 Associating lights with a surface

For efficiency reasons, the shading system must know which light shaders will be used with a surface shader before the surface shader is compiled.

`sglUseLight(GLuint lightShaderID)`

This command binds a "compiled" light shader to the current surface-shader source code. `lightShaderID` indicates the light that is to be associated with the surface.

More than one light can be associated with a surface, by calling `sglUseLight()` multiple times.

However, the same (compiled) light shader may not be used more than once with a single surface. If two identical lights are required, compile the light shader twice. Our system imposes this requirement because the *lightShaderID* is used to specify how light parameters are modified. "Identical" lights will usually have different parameter values (e.g. position).

5.1 Setting parameter values

For performance reasons, shader parameters are identified at rendering-time with numeric identifiers rather than names. For each compiled shader, the programmer can choose the bindings from names to numeric identifiers, within some constraints. We refer to the numeric parameter identifiers as *parameter handles*. Each compiled surface or light shader has its own parameter-handle space.

There is an important advantage to allowing the programmer to choose values of parameter handles. It facilitates the use of a single geometry rendering routine (e.g. renderSphere) with different surface shaders, as long as the programmer chooses a consistent mapping of parameter handles to actual parameters for all of the relevant shaders.

`sglParameterHandle(char *paramName, GLuint paramHandle)`

Assigns the parameter handle `paramHandle` to the current shader's parameter `paramName`. The value of paramHandle must be between 0 and `SGL_MAX_PARAMHANDLE`. The value of `SGL_MAX_PARAMHANDLE` is guaranteed to be no less than 15.

`sglParameter*(GLuint paramHandle, TYPE v, ...)`

`sglParameter*v(GLuint paramHandle, TYPE *v)`

Assigns a value to the shader parameter(s) specified by `paramHandle`. For a per-vertex parameter, this routine may be called at any time. For a per-primitive parameter, this routine may only be called outside of a begin/end pair.

Because our shading language does not explicitly identify shader parameters as "colors" or "texture coordinates", the shading system can not automatically assign default values in the manner that OpenGL does. For example, in OpenGL a `glColor3f` command automatically sets the fourth value (alpha) to the default

value of 1.0. When using our system, the user must always specify all four components of the color value. Likewise, the user must always specify all four components of a texture value. For a 2D texture, the third and fourth values should usually be set to 0.0 and 1.0 respectively.

The `sglParameter*` routine is available in `sglParameter1*`, `sglParameter4*`, and `sglParameter16*` variants. The `sglParameter16*` variants are used to specify matrix parameters, using OpenGL's array format.

If the shading language specifies a parameter's type as either `clampf` or `clampfv`, type conversions are performed in the same manner as they are for the OpenGL `glColor*` routines (see OpenGL Red Book, 3rd edition, Table 4-1).[1]. In summary, integer-to-float conversions are performed such that the maximum integer value (e.g. 255 for an unsigned byte) maps to 1.0. This behavior allows colors and normals to be stored in unsigned bytes in a natural manner.

Our shading language uses textures, but the contents of the textures are not defined using the language. Textures are defined by the application program, then passed to the shading-language routine as a 'texref' parameter. Our system relies on OpenGL's texture object facility (`glBindTexture()`). The `sglParameter1i` or `sglParameter1iv` routines are used to specify 'texref' parameters. The value of the integer parameter is the *textureName* created using `glBindTexture()`.

```
sglLightParameter*(GLuint lightShaderID, GLuint paramHandle, TYPE v, ...)
sglLightParameter*v(GLuint lightShaderID, GLuint paramHandle, TYPE *v)
```

Assigns a value to the light parameter specified by `paramHandle`. The "compiled" light shader is specified by `lightShaderID`. For a per-vertex parameter, this routine may be called at any time. For a per-primitive parameter, this routine may only be called outside of a begin/end pair.

5.2 Light Pose

The pose of a light (position, direction, and orientation) is set using a routine defined for that purpose.

```
sglLightPosefv(GLuint lightShaderID, GLuint pname, GLfloat *v)
```

`pname` can be `SGL_POSITION`, `SGL_DIRECTION`, or `SGL_UPAXIS`. The light direction defines the $-Z$ axis in light space, and the up axis defines the Y axis in light space.

The vector `v` should always be a four-element vector, and is considered to be in modelview space (i.e. transformed by the modelview matrix or its inverse transpose, as appropriate). For `SGL_POSITION`, the fourth element of the vector should usually be set to 1.0. For `SGL_DIRECTION` and `SGL_UPAXIS`, the fourth element should usually be set to 0.0.

[1] For implementation simplicity, our system deviates from the behavior in Table 4-1 in a minor way. Our system treats negative and positive values symmetrically. For example, a signed-byte value of -127 maps to -1.0, whereas in OpenGL the value of -128 maps to -1.0

5.3 Ambient Light

`sglAmbient*(...)`

Specify the global ambient color. This color is accessible in surface shaders using the pre-defined `Ca` variable. If a surface shader does not use the `Ca` variable, the ambient color will be ignored. This routine can not be called inside a Begin/End pair.

6 Replacements for Standard OpenGL routines

6.1 Begin/End and Flush/Finish

Use `sglBegin()`, `sglEnd()`, `sglFlush()`, and `sglFinish()` instead of the corresponding standard OpenGL routines. Using the standard OpenGL routines while a programmable shader is active will result in undefined behavior.

6.2 Vertices, Normals, Tangents, Binormals

```
sglVertex*(TYPE v, ...)
sglVertex*v(TYPE v, ...)
sglNormal3*(TYPE v, ...)
sglNormal3*v(TYPE v, ...)
sglTangent3*(TYPE v, ...)
sglTangent3*v(TYPE v, ...)
sglBinormal3*(TYPE v, ...)
sglBinormal3*v(TYPE v, ...)
```

Vertices and local coordinate-frame vectors are passed using our versions of the classical OpenGL routines. The results of calling one of the standard OpenGL routines while a programmable shader is active are undefined.

6.3 Vertex arrays

To attain higher frame rates when using large models, the shading system provides `sglParameterPointer`, `sglEnableClientState`, `sglGetClientState`, `sglDisableClientState` and `sglDrawArrays`. These routines differ from the OpenGL routines in that they support not only arrays of vertices, normals, binormals or tangents, but also of any other shader parameter. To setup vertex arrays, you have to follow a basic three step procedure, consisting of calls to:

1. `sglParameterPointer`
2. `sglEnableClientState`
3. `sglDrawArrays`.

First, the pointers to the parameter arrays have to be specified by `sglParameterPointer(int handle, GLsizei size, GLenum type, GLsizei stride, float *pointer)`. Valid handles are `SGL_VERTEX`, `SGL_NOMRAL`, `SGL_BINORMAL`, `SGL_TANGENT` or any parameter handle obtained from `sglParameterHandle`. The parameters `size`, `type`, `stride` and `pointer` follow standard OpenGL vertex-array conventions. Please note that in the current version of the immediate-mode API:

- GL_FLOAT is the only supported type.
- Stride should always be set to 4 for `SGL_VERTEX`, and to 3 for `SGL_NORMAL`, `SGL_BINORMAL` or `SGL_TANGENT` arrays.

After specifying all parameter arrays, they must be activated for rendering by calling `sglEnableClientState(int handle)`. Similar, an activated parameter array can be disabled again by calling `sglDisableClientState(int handle)`.

To render the actual vertex array using all activated parameter arrays, call `sglDrawArrays(GLenum mode, GLint first, GLsizei count)` which again follows standard OpenGL conventions. Please note that rendering will only occur if an SGL_VERTEX array was both specified and activated. All other parameter arrays are optional. `SGL_NOMRAL`, `SGL_BINORMAL` and `SGL_TANGENT` are set to constant default values if not provided.

7 Advanced features

7.1 Manual backend configuration

To offer manual control over which backends the shading system should use, the immediate-mode interface provides `sglSetBackEndType(char* perprimitivegroup, char* vertex, char* fragment)`. This routine presents a wraper for the internal, low-level functions `set_bcodegen`, `set_vcodegen` and `set_fcodegen`.

Currently, there are two primitive-group backends ("cc" and "x86"), three vertex backends ("cc", "x86", and "nv20"), and two fragment backends ("lb" and "nv"). "lb" is a standard-OpenGL backend; "nv" is a register-combiner backend.

7.2 Shader parameter list retrieval

The following two routines allow a program to retrieve the lists of parameters required by a surface shader. To retrieve the number of parameters for the current surface shader:
`sglGetParameterCount(int *count)` where count returns the total number of parameters for the shader.

To retrieve the name and number of values for a specific parameter:
`sglGetParameterInfo(int p, char **name, int *vcnt)` where p defines the parameter of the current shader, ranging from 0 to (1-count), name returns the name of the parameter and vcnt returns the number of values for this parameter. E.g. for a float4 parameter, vcnt would return 4.

To retrieve the lists of parameters required by a light shader, use the following routines:
`sglGetLightParameterCount(int lightid, int *count)`
`sglGetLightParameterInfo(int lightid, int p, char **name, int *vcnt)`
Both routines take a `lightid` as parameter that specifies the light for which information should be retrieved.

8 Depth testing

Ideally, depth testing works exactly as it does in standard OpenGL. However, in some implementations, incorrect shading may occur if two (potentially visible) fragments at a pixel have exactly the same depth. This problem only occurs if an implementation uses the framebuffer for inter-pass temporary storage in a multi-pass shader.

9 Error Handling

The shading system has a flexible method for handling errors. Errors are divided into two classes, minor and major. For each class of error, the application can choose one of four behaviors:

- SG_MSG_NONE – No message is printed, and program execution continues. Errors can only be detected by polling for them using sglGetError.
- SGL_MSG_WARN_ONCE – A message is printed for the first error that occurs, and program execution continues. No message is printed for subsequent errors.
- SGL_MSG_WARN – A message is printed for every error that occurs, and program execution continues.
- SGL_MSG_ABORT – When an error occurs, a messsage is printed and program execution is halted.

```
sglDebugLevel(int minor, int major)
```

Specify the behavior for minor and major errors. The default is sglDebugLevel(SGL_MSG_WARN_ONCE, SGL_MSG_ABORT).

```
GLenum sglGetError(void)
```

Poll for an error. If no error has occurred, GL_NO_ERROR is returned. If an error has occurred, the error code is returned.

```
const GLubyte* sglErrorString(GLenum errorCode)
```

Returns a descriptive string corresponding to an error code.

10 System Tips

In our current implementation, every sglBegin/sglEnd pair is expensive. If possible, group all primitives into one such pair.

Because of restrictions in current graphics hardware, if a translucent shader is implemented using more than one hardware pass, overlapping transparent primitives will not render correctly. You must call sglFlush between each group of potentially overlapping primitives to avoid this problem.

Simple Program that uses Immediate-Mode Interface

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <GL/glut.h>

#include "imode.h"

/*
 * Macro to check OpenGL error status, and print message if so
 */
#define check_gl_error do {GLenum glerr; \
    while ((glerr = glGetError()) != GL_NO_ERROR) \
      fprintf(stderr, "OpenGL error '%s' at %s:%i\n", gluErrorString(glerr), \
                      __FILE__, __LINE__);} while(0)

/* The following parameter handles can be chosen mostly arbitrarily
   (must be smallish, unique numbers) */
#define PH_COLOR      1
#define PH_AC         2
#define PH_AL         3
#define PH_AQ         4
#define PH_TEX        7
#define PH_UV         8
#define PH_SURFCOLOR  9

/* glBindTexture ID */
#define TEXID 1

GLfloat light_diffuse[] = {1.0, 0.0, 0.0, 1.0};

/*
 * Checkboard texture
 */
#define checkWidth  64
#define checkHeight 64
static GLubyte checkImage[checkWidth][checkHeight][4];

void makeCheckImage(void) {
  int i,j,c;
  for (i=0; i<checkHeight; i++) {
    for (j=0; j<checkWidth; j++) {
      c = (((i&0x8)==0)^((j&0x8)==0))*255;
      checkImage[i][j][0] = (GLubyte) c;
      checkImage[i][j][1] = (GLubyte) c;
      checkImage[i][j][2] = (GLubyte) c;
      checkImage[i][j][3] = (GLubyte) 255;
    }
  }
}

/*
 * Sets up checkboard as texture #TEXID
 */
void setupTexture() {
  makeCheckImage();
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glBindTexture(GL_TEXTURE_2D, TEXID);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                  GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkWidth, checkHeight,
               0, GL_RGBA, GL_UNSIGNED_BYTE, checkImage);
}

/*
 * draw cube with clockwise verts when looking at cube from outside
 */
void drawcube(void) {
  GLfloat red[]   = {1.0, 0.0, 0.0, 1.0};
  GLfloat green[] = {0.0, 1.0, 0.0, 1.0};
  GLfloat blue[]  = {0.0, 0.0, 1.0, 1.0};
  float UVa[] = {0.0, 0.0, 0.0, 1.0};
  float UVb[] = {0.0, 1.0, 0.0, 1.0};
  float UVc[] = {1.0, 1.0, 0.0, 1.0};
  float UVd[] = {1.0, 0.0, 0.0, 1.0};

  sglBegin(GL_QUADS);
  sglParameter4fv(PH_SURFCOLOR, red);
  /* x=1 face */
  sglNormal3f( 1.0,  0.0,  0.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f( 1.0, -1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f( 1.0, -1.0,  1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0,  1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f( 1.0,  1.0, -1.0);
  /* x=-1 face */
  sglNormal3f(-1.0,  0.0,  0.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0, -1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f(-1.0,  1.0, -1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f(-1.0,  1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f(-1.0, -1.0,  1.0);
  /* y=1 face */
  sglParameter4fv(PH_SURFCOLOR, green);
  sglNormal3f( 0.0,  1.0,  0.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0, 1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f( 1.0, 1.0, -1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0, 1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f(-1.0, 1.0,  1.0);
  /* y=-1 face */
  sglNormal3f( 0.0,-1.0,  0.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0,-1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f(-1.0,-1.0,  1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0,-1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f( 1.0,-1.0, -1.0);
  /* z=1 face */
  sglParameter4fv(PH_SURFCOLOR, blue);
  sglNormal3f( 0.0, 0.0,  1.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0,-1.0,  1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f(-1.0, 1.0,  1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0, 1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f( 1.0,-1.0,  1.0);
  /* z=-1 face */
  sglNormal3f( 0.0, 0.0, -1.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0,-1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f( 1.0,-1.0, -1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0, 1.0, -1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f(-1.0, 1.0, -1.0);
  sglEnd();
}
```

continued on next page

```
static void init_shader_params(void) {
  static float light_ambient[4] = { 0.2, 0.2, 0.2, 1.0 };
  static int frame = 0;

  /*
   * Changes to modelview matrix
   */
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 5,    /* Eye */
                    0.0, 0.0, 0.0,  /* Center */
                    0.0, 1.0, 0.0); /* Up */
  glTranslatef(0.0, 0.0, -1.0);
  glRotatef((float)frame++, 0.0, 1.0, 0.0);
  glRotatef(35.264, 1.0, 0.0, 0.0);
  glRotatef(45, 0.0, 0.0, 1.0);

  sglAmbient4fv(light_ambient);
}

/* GLUT keyboard callback -- Quit when 'Q' key is pressed */
void keyboard(unsigned char key, int x, int y) {
  switch (key) {
  case 'q': case 'Q': /* quit */
    exit(0);
    break;
  }
}

/* GLUT reshape callback -- reset viewport when window size changes */
void reshape(GLint w, GLint h) {
  sglViewport(0, 0, w, h);
}

/* GLUT idle callback -- continuously redraw so that we get animation */
void dynamicIdle(void) {
  glutPostRedisplay();
}

/* GLUT display callback -- draw the scene */
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  sglBindShader(200);
  init_shader_params();
  drawcube();
  glutSwapBuffers();
  check_gl_error;
}

void gfxinit(void) {
  float light_color[4] = { 1.0, 1.0, 1.0, 1.0 };
  float atten_constant = 1.0;
  float atten_linear = 0.01;
  float atten_quadratic = 0.0;
  GLfloat light_position[] = {3.0, 3.0, 3.0, 1.0};

  glEnable(GL_DEPTH_TEST);
  glMatrixMode(GL_PROJECTION);
  gluPerspective( /* FOV in deg */ 40.0, /* Aspect ratio */ 1.0,
                  /* Znear */ 1.0, /* Zfar */ 10.0);
  glMatrixMode(GL_MODELVIEW);
  gluLookAt(0.0, 0.0, 5,    /* Eye */
                    0.0, 0.0, 0.0,  /* Center */
                    0.0, 1.0, 0.0); /* Up */
  glTranslatef(0.0, 0.0, -1.0);

  /*
   * Shading system setup
   */
#if 0
  /* Specify specific codegens; without this defaults are used */
  set_bcodegen("x86");
  set_vcodegen("nv20");
  set_fcodegen("nv");
#endif
  sglInit();


  /*
   * Load and compile light shader
   */
  sglShaderFile(99, "simple_light", "../simpshade.in");
  sglCompileShader(299);
  sglParameterHandle("color",       PH_COLOR);
  sglParameterHandle("ac",          PH_AC);
  sglParameterHandle("al",          PH_AL);
  sglParameterHandle("aq",          PH_AQ);
  /*
   * Specify light position & configuration
   */
  sglLightPosefv(299,               SGL_POSITION,     light_position);
  sglLightParameter4fv(299,         PH_COLOR,         light_color);
  sglLightParameter1f(299,          PH_AC,            atten_constant);
  sglLightParameter1f(299,          PH_AL,            atten_linear);
  sglLightParameter1f(299,          PH_AQ,            atten_quadratic);

  /*
   * Load and compile surface shader
   */
  sglShaderFile(1, "simple_surface", "../simpshade.in");
  sglUseLight(299);
  sglCompileShader(200);
  sglParameterHandle("surfcolor",   PH_SURFCOLOR);
  sglParameterHandle("tex",         PH_TEX);
  sglParameterHandle("uv",          PH_UV);
}

int main(int argc, char **argv) {
  /*
   * GLUT setup
   */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutCreateWindow("simple");
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutIdleFunc(dynamicIdle);
  glutKeyboardFunc(keyboard);

  /*
   * Initialize graphics
   */
  gfxinit();
  setupTexture();
  sglParameter1i(PH_TEX, TEXID);

  /*
   * Start event loop
   */
  glutMainLoop();
  return 0;
}
```

Chapter 9

Sampling Procedural Shaders

Wolfgang Heidrich

Real-time Shading: Sampling Procedural Shaders

Wolfgang Heidrich
The University of British Columbia

Abstract

In interactive or real-time applications, naturally the complexity of tasks that can be performed on the fly is limited. For that reason it is likely that even with the current rate of development in graphics hardware, the more complex shaders will not be feasible in this kind of application for some time to come.

One solution to this problem seems to be a precomputation approach, where the procedural shader is evaluated (sampled), and the resulting values are stored in texture maps, which can then be applied in interactive rendering. A closer look, however, reveals several technical difficulties with this approach. These will be discussed in this section, and hints towards possible solutions will be given.

1 Introduction and Problem Statement

In order to come up with an approach for sampling procedural shaders, we first have to determine which aspects of the shading system we would like to alter in the interactive application.

For example, we can evaluate the shader for a number of surface locations with fixed illumination (all light source positions and parameters are fixed), and a fixed camera position. This is the mode in which a normal procedural shading system would evaluate the shader for a surface in any given frame. If the shader is applied to a parametric surface $F(u, v)$, then we can evaluate the shader at a number of discrete points (u, v), and store the resulting color values in a texture map.

In an interactive application, however, this particular example is of limited use since both the viewer and the illumination is fixed. As a result the texture can only be used for exactly one frame, unless the material is completely diffuse. In a more interesting scenario, only the illumination is fixed, but the camera is free to move around in the scene. In this case, the shader needs to be evaluated for many different reference viewing positions, and during realtime rendering the texture for any given viewing direction can be interpolated from the reference images. This four-dimensional data structure (2 dimensions for u and v, and 2 dimensions for the camera position) is called a light field, and is briefly described in Section 4.

If we want to go one step further, and keep the illumination flexible as well, we end up with a even higher dimensional data structure. There are several ways to do this, but one of the more promising is probably the use of a *space-variant BRDF*, i.e. a reflection model whose parameters can change over a surface. This yields an approach with a six-dimensional data structure that will be outlined in Section 5.

No matter which of these approaches is to be taken, there are some issues that have to be resolved for all of them. One of them is the choice of an appropriate resolution for the sampling process. The best resolution depends on many different factors, some of which depend on the system (i.e. the amount of memory available, or the range of viewing distances under which the shaded object will be seen), and some of which depend on the shader (i.e. the amount of detail generated by the shader).

In the case of a 2D texture with fixed camera and lighting, a sample resolution can still be chosen relatively easily, for example, by letting the user make a decision. With complex view-dependent effects this is much harder because it is hard to determine appropriate resolutions for sampling specular highlights whose sharpness may vary over the surface. An automatic method for estimating the resolution would be highly desirable.

Another problem is the sheer number of samples that we may have to acquire. For example, to sample a shader as a space variant BRDF with a resolution of

$256x256$ for the surface parameters u and v, as well as 32^2 samples for both the light direction and the viewing direction requires over 68 billion samples, which is unfeasible both in terms of memory consumption and the time required to acquire these samples. On the other hand, it is to be expected that the shader function is relatively smooth, with the high-frequency detail localized in certain combinations of viewing and light directions (specular highlights, for example). Thus, a hierarchical sampling scheme is desirable which allows us to refine the sampling in areas that are more complex without having to do a high-density sampling in all areas. At the same time the hierarchical method should make sure we do not miss out on any important features. Such an approach is described in the next section.

2 Area Sampling of Procedural Shaders

In this section we introduce the concept of *area sampling* a procedural shader using a a certain kind of arithmetic that replaces the standard floating point arithmetic. This *affine arithmetic* allows us to evaluate a shader over a whole area, yielding an upper and a lower bound for all the values that the shader takes on over this area. This bound can then be used hierarchically to refine the sampling in areas in which the upper and the lower bound are far apart (i.e. areas with a lot of detail). The full details of the method can be found in [10].

We will discuss the general approach in terms of sampling a 2D texture by evaluating a shader with a fixed camera position and illumination. The same methods can however be applied to adaptively adjust the sampling rates for camera and light position.

2.1 Affine Arithmetic

Affine arithmetic (AA), first introduced in [4], is an extension of interval arithmetic [16]. It has been successfully applied to several problems for which interval arithmetic had been used before [17, 20, 21]. This includes reliable intersection tests of rays with implicit surfaces, and recursive enumerations of implicit surfaces in quad-tree like structures [5, 6].

Like interval arithmetic, AA can be used to manipulate imprecise values, and to evaluate functions over intervals. It is also possible to keep track of truncation and round-off errors. In contrast to interval arithmetic, AA also maintains dependencies between the sources of error, and thus manages to compute significantly tighter error bounds. Detailed comparisons between interval arithmetic and affine arithmetic can be found in [4], [5], and [6].

Affine arithmetic operates on quantities known as *affine forms*, given as polynomials of degree one in a set of *noise symbols* ϵ_i.

$$\hat{x} = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \cdots + x_n\epsilon_n$$

The coefficients x_i are known real values, while the values of the noise symbols are unknown, but limited to the interval $\mathbf{U} := [-1, 1]$. Thus, if all noise symbols can independently vary between -1 and 1, the range of possible values of an affine form $\hat{x}$ is

$$[\hat{x}] = [x_0 - \xi, x_0 + \xi], \quad \xi = \sum_{i=1}^{n} |x_i|.$$

Computing with affine forms is a matter of replacing each elementary operation $f(x)$ on real numbers with an analogous operation $f^*(\epsilon_1, \ldots, \epsilon_n) := f(\hat{x})$ on affine forms.

If f is itself an affine function of its arguments, we can apply normal polynomial arithmetic to find the corresponding operation f^*. For example we get

$$\begin{aligned} \hat{x} + \hat{y} = &\quad (x_0 + y_0) + (x_1 + y_1)\epsilon_1 + \cdots + (x_n + y_n)\epsilon_n \\ \hat{x} + \alpha = &\quad (x_0 + \alpha) + x_1\epsilon_1 + \cdots + x_n\epsilon_n \\ \alpha\hat{x} = &\quad \alpha x_0 + \alpha x_1\epsilon_1 + \cdots + \alpha x_n\epsilon_n \end{aligned}$$

for affine forms $\hat{x}, \hat{y}$ and real values α.

3 Non-Affine Operations

If f is not an affine operation, the corresponding function $f^*(\epsilon_1, \ldots, \epsilon_n)$ cannot be exactly represented as a linear polynomial in the ϵ_i. In this case it is necessary to find an affine function $f^a(\epsilon_1, \ldots, \epsilon_n) = z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n$ approximating $f^*(\epsilon_1, \ldots, \epsilon_n)$ as well as possible over $\mathbf{U}^n$. An additional *new* noise symbol ϵ_k has to be added to represent the error introduced by this approximation. This yields the following affine form for the operation $z = f(x)$:

$$\hat{z} = f^a(\epsilon_1, \ldots, \epsilon_n) = z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n + z_k\epsilon_k,$$

with $k \notin \{1, \ldots, n\}$. The coefficient z_k of the new noise symbol has to be an upper bound for the error introduced by the approximation of f^* with f^a:

$$z_k \geq \max\{|f^*(\epsilon_1, \ldots, \epsilon_n) - f^a(\epsilon_1, \ldots, \epsilon_n)| : \epsilon_i \in \mathbf{U}\}.$$

For example it turns out (see [4] for details) that a good approximation for the multiplication of two affine forms $\hat{x}$ and $\hat{y}$ is

$$\hat{z} = x_0y_0+(x_0y_1+y_0x_1)\epsilon_1+\cdots+(x_0y_n+y_0x_n)\epsilon_n+uv\epsilon_k,$$

with $u = \sum_{i=1}^{n} |x_i|$ and $v = \sum_{i=1}^{n} |y_i|$. In general, the best approximation f^a of f^* minimizes the Chebyshev error between the two functions.

The generation of affine approximations for most of the functions in the standard math library is relatively straightforward. For a univariate function $f(x)$, the iso-surfaces of $f^*(\epsilon_1, \ldots, \epsilon_n) = f(x_0 + x_1\epsilon_1 + \cdots + x_n\epsilon_n)$ are hyperplanes of $\mathbf{U}^n$ that are perpendicular to the vector $(x_1, \ldots, x_n)$. Since the iso-surfaces of every affine function $f^a(\epsilon_1, \ldots, \epsilon_n) = z_0 + z_1\epsilon_1 + \cdots + z_n\epsilon_n$ are also hyperplanes of this space, it is clear that the iso-surfaces of the best affine approximation f^a of f^* are also perpendicular to $(x_1, \ldots, x_n)$. Thus, we have

$$f^a(\epsilon_1, \ldots, \epsilon_n) = \alpha\hat{x}+\beta = \alpha(x_0+x_1\epsilon_1+\cdots+x_n\epsilon_n)+\beta$$

for some constants α and β. As a consequence, the minimum of $\max_{\epsilon_i \in \mathbf{U}} |f^a - f^*|$ is obtained by minimizing $\max_{\epsilon_i \in \mathbf{U}} |f(\hat{x}) - \alpha\hat{x} - \beta| = \max_{x \in [a,b]} |f(x) - \alpha x - \beta|$, where $[a, b]$ is the interval $[\hat{x}]$. Thus, approximating f^* has been reduced to finding a linear Chebyshev approximation for a univariate function, which is a well understood problem [3]. For a more detaile discussion, see [10].

Most multivariate functions can handled by reducing them to a composition of univariate functions. For example, the maximum of two numbers can be rewritten as $\max(x, y) = \max_0(x - y) + y$, with $\max_0(z) := \max(z, 0)$. For the univariate function $\max_0(z)$ we can use the above scheme.

3.1 Application to Procedural Shaders

In order to apply AA to procedural shaders, it is necessary to investigate which additional features are provided by shading languages in comparison to standard math libraries. In the following, we will restrict ourselves to the functionality of the RenderMan shading language [9, 18, 22], which is generally agreed to be one of the most flexible languages for procedural shaders. Since its features are a superset of most other shading languages (for example [2] and [15]), the described concepts apply to these other languages as well.

Shading languages usually introduce a set of specific data types and functions exceeding the functionality of general purpose languages and libraries. Most of these additional functions can easily be approximated by affine forms using techniques similar to the ones outlined in the previous section. Examples for this kind of domain specific functions are continuous and discontinuous transitions between two values, like step functions, clamping of a value to an interval, or smooth Hermite interpolation between two values.

The more complicated features include splines, pseudo-random noise, and derivatives of expressions. For an in-depth discussion of these functions we refer the reader to the original paper [10].

New data types in the RenderMan shading language are points and color values, both simply being vectors of scalar values. Affine approximations of the typical operations on these data types (sum, difference, scalar-, dot- and cross product, as well as the vector norm) can easily be implemented based on the primitive operations on affine forms.

Every shader in the RenderMan shading language is supplied with a set of explicit, shader specific parameters that may be linearly interpolated over the surface, as well as fixed set of implicit parameters (global variables). The implicit parameters include the location of the sample point, the normal and tangents in this point, as well as vectors pointing towards the eye and the light sources. For parametric surfaces, these values are functions of the surface parameters u and v, as well as the size of the sample region in the parameter domain: du and dv.

For parametric surfaces including all geometric primitives defined by the RenderMan standard, the explicit and implicit shader parameters can therefore be computed by evaluating the corresponding function over the affine forms for u, v, du, and dv. The affine forms of these four values have to be computed from the sample region in parameter space. For many applications, du and dv will actually be real values on which the affine forms of u and v depend: $\hat{u} = u_0 + du \cdot \epsilon_1$ and $\hat{v} = v_0 + dv \cdot \epsilon_2$.

With this information, we can set up a hierarchical sampling scheme as follows. The shader is first evaluated over the whole parameter domain ($u = 0.5 + 0.5 \cdot \epsilon_1, v = 0.5 + 0.5 \cdot \epsilon_2$). If the resulting upper and lower bound of the shader are too different, the parameter domain is hierarchically subdivided into four regions, and area samples for these regions are computed. The re-

cursion stops when the difference between upper and lower bound (error) is below a certain limit, or if the maximum subdivision level is reached. Results of this approach together with an error plot are given in Figure 1.

3.2 Analysis

In our description we uses affine arithmetic to obtain conservative bounds for shader values over a parameter range. In principle, we could also use any other range analysis method for this purpose. It is, however, important that the method generates tight, conservative bounds for the shader. Conservative bounds are important to not miss any small detail, while tight bounds reduce the number of subdivisions, and therefore save both computation time and memory.

We have performed tests to compare interval arithmetic to affine arithmetic for the specific application of procedural shaders. Our results show that the bounds produced by interval arithmetic are significantly wider than the bounds produced by affine arithmetic. Figure 2 shows the wood shader sampled at a resolution of 512×512. The error plots show that interval arithmetic yields errors up to 50% in areas where affine arithmetic produces errors below $1/256$. As a consequence, the textures generated from this data by assigning the mean values of the computed range to each pixel, reveal severe artifacts in the case of interval arithmetic.

The corresponding error histogram in Figure 3 shows that, while the most of the per-pixel errors for affine arithmetic are are less than 3%, most of the errors for interval arithmetic are in the range of 5%-10%, and a significant number is even higher than this (up to 50%).

These results are not surprising. All the expressions computed by a procedural shader depend on four input parameters: u, v, du, and dv. Affine arithmetic keeps track of most of these subtle dependencies, while interval arithmetic ignores them completely. The more complicated functions get, the more dependencies between the sources of error exist, and the bigger the advantage of AA. These results are consistent with prior studies published in [4], [5], and [6].

The bounds of both affine and interval arithmetic can be further improved by finding optimal approximations for larger blocks of code, instead of just library functions. This process, however, requires human intervention and cannot be done automatically.

This leaves us with the method presented here as the only practical choice, as long as conservative error bounds are required. Other applications, for which an estimate of the bounds is sufficient, could also use Monte Carlo sampling. In this case it is interesting to analyze the number of Monte Carlo samples and the resulting quality of the estimate that can be obtained in the same time as a single area sample using AA. Table 1 shows a comparison of these numbers in terms of floating point operations (FLOPS) and execution time (on a 100MHz R4000 Indigo) for the various shaders used throughout this paper.

For more complicated shaders the relative performance of AA decreases, since more error variables are introduced due to the increased amount of non-affine operations. The table shows that, depending on the shader, 5 to 10 point samples are as expensive as a single AA area sample. To see what this means for the quality of the bounds, consider the screen shader with a density of 0.5. The density of 0.5 means that 75 percent of the shader will be opaque, while 25 percent will be translucent. If we take 7 point samples of this shader, which is about as expensive as a single AA sample, the probability that all samples compute the same opacity is $0.75^7 + 0.25^7 \approx 13.4$ percent. Even with 10 samples the probability is still 5.6 percent.

For the example of using area samples as a subdivision criterion in hierarchical radiosity, this means that a wall covered with the screen shader would have a probability of 13.4 (or 5.6) percent of not being subdivided at all. The same probability applies to each level in the subdivision hierarchy independently. These numbers indicate that AA is superior to point sampling even if only coarse estimates of the error bounds are desired.

4 Light Fields

Let us now consider how the method can be used in a scenario with a varying camera location, but fixed illumination. This is somewhat speculative, because it has never actually been tried. It is therefore to be expected that in practical implementations some new issues will arise that will have to be resolved in future research.

Before we outline an approach for adaptively acquiring light fields from procedural shaders, we will first review the concept of a light field itself.

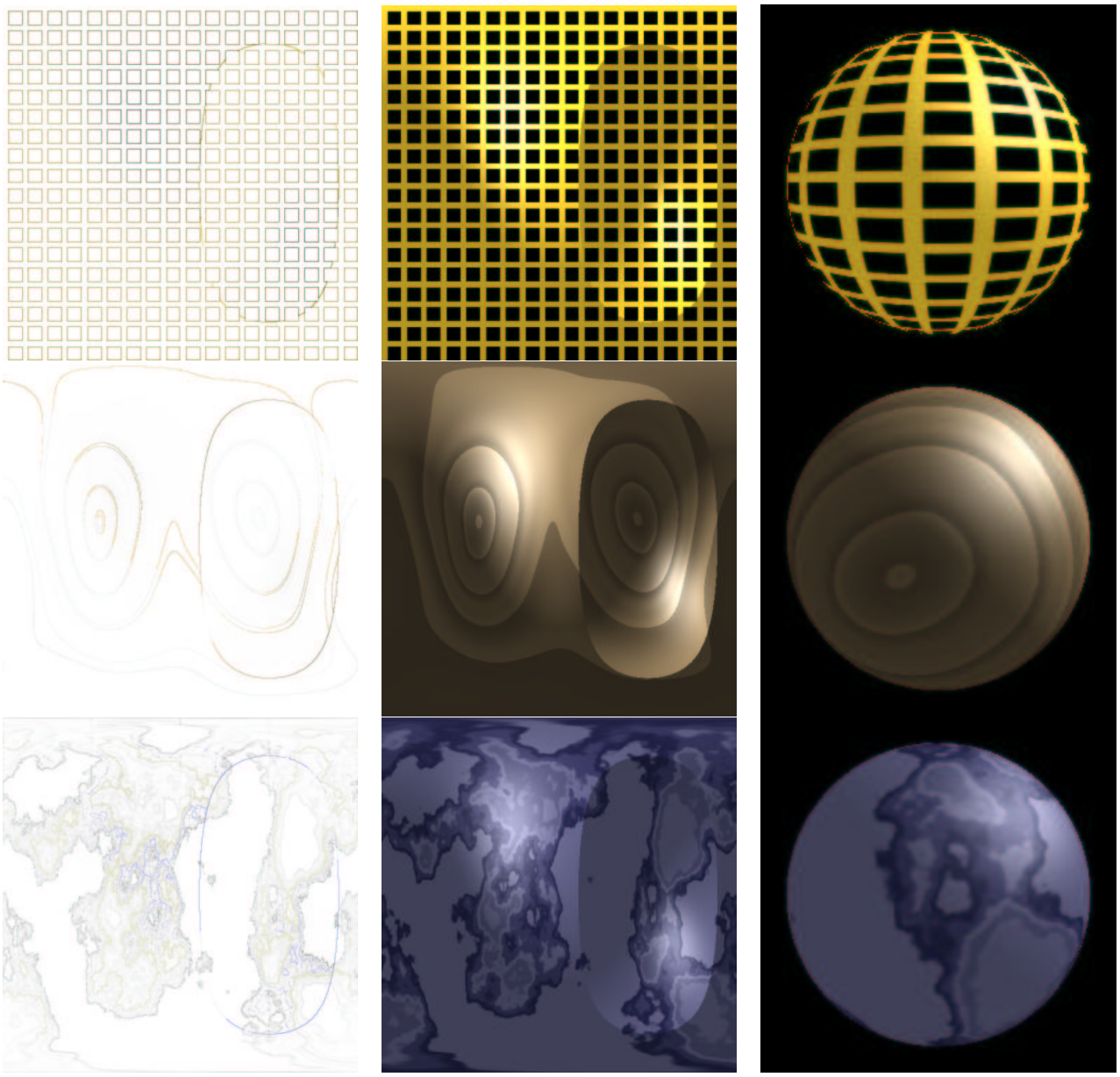


Figure 1: Several examples of RenderManshaders samples with affine arithmetic. Left: per-pixel error bounds, center: generated texture, right: texture applied to 3D geometry.

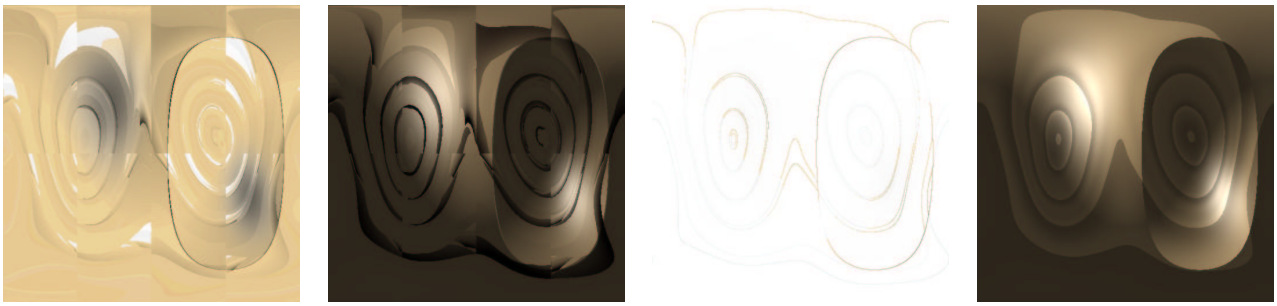


Figure 2: The wood shader sampled at a resolution of 512×512. From left to right: error plot using interval arithmetic, resulting texture, error plot using affine arithmetic, resulting texture.

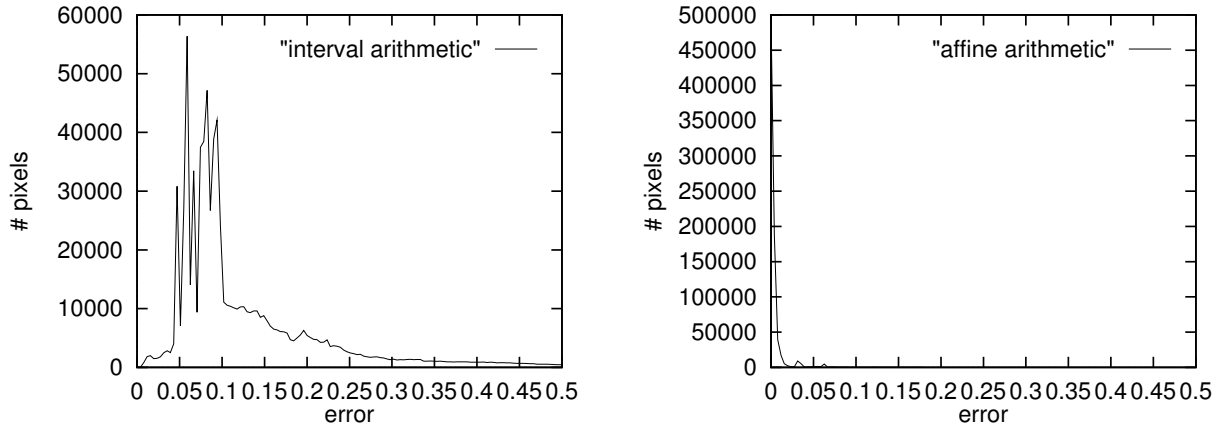


Figure 3: Error histograms for the wood shader for interval arithmetic (left) and affine arithmetic (right).

Shader	FLOPS (ps)	FLOPS (aa)	ratio	Time (ps)	Time (aa)	ratio
screen	24	214	1:8.92	4.57	33.48	1:7.32
wood	803	6738	1:8.39	8.34	86.53	1:10.38
marble	4386	28812	1:6.57	9.46	88.52	1:9.36
bumpmap	59	487	1:8.25	3.76	20.43	1:5.43
eroded	2995	26984	1:9.01	18.85	193.33	1:10.27

Table 1: FLOPS per sample and timings for 4096 samples, for stochastic point sampling (ps) and AA area sampling (aa).

4.1 Definition

A light field[13] is a 5-dimensional function describing the radiance at every point in space in each direction. It is closely related to the *plenoptic function* introduced by Adelson[1], which in addition to location and orientation also describes the wavelength dependency of light.

In the case of a scene that is only to be viewed from outside a convex hull, it is sufficient to know what radiance leaves each point on the surface of this convex hull in any given direction. Since the space outside the convex is assumed to be empty, and radiance does not change along a ray in empty space, the dimensionality of the light field can be reduced by one, if an appropriate parameterization is found. The so-called two-plane parameterization fulfills this requirement. It represents a ray via its intersection points with two parallel planes. Several of these pairs of planes (also called *slabs*) are required to represent a complete hull of the object. Since each of these points is characterized by two parameters in the plane, this results in a 4-dimensional function that can be densely sampled through a regular grid on each plane (see Figure 4).

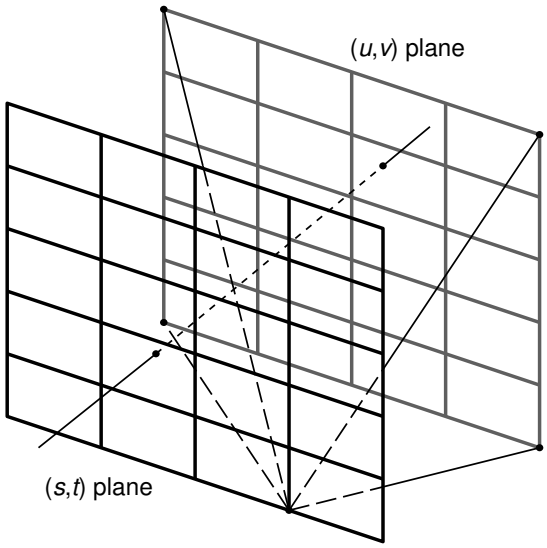


Figure 4: A light field is a 2-dimensional array of images taken from a regular grid of eye points on the (s,t)-plane through a window on the (u,v)-plane. The two planes are parallel, and the window is the same for all eye points.

One useful property of the two-plane parameterization is that all the rays passing through a single point on the (s,t)-plane form a perspective image of the scene,

with the (s,t) point being the center of projection. Thus, a light field can be considered a 2-dimensional array of perspective projections with eye points regularly spaced on the (s,t)-plane. Other properties of this parameterization have been discussed in detail by Gu et al.[8].

Since we assume that the sampling is dense, the radiance along an arbitrary ray passing through the two planes can be interpolated from the known radiance values in nearby grid points. Each such ray passes through one of the grid cells on the (s,t)-plane and one on the (u,v)-plane. These are bounded by four grid points on the respective plane, and the radiance from any of the (u,v)-points to any of the (s,t)-points is stored in the data structure. This makes for a total of 16 radiance values, from which the radiance along the ray can be interpolated quadri-linearly. As shown in by Gortler et al[7] and Sloan et al.[19], this algorithm can be considerably sped up by the use of texture mapping hardware. Sloan et al.[19] also propose a generalized version of the two-plane parameterization, in which the eye points can be distributed unevenly on the (s,t)-plane, while the samples on the (u,v)-plane remain on a regular grid.

A related data structure is the *surface light field* [14, 23], in which two of the four parameters of the light field are attached to the surface parameters. That is, u and v correspond to the parameters of a parametric surface, while s and t specify the viewing direction. The details of the different variants of surface light fields are beyond the scope of this document, and we refer the interested reader to the original papers [14, 23].

4.2 Sampling of Light Fields

The sampling method from Section 3.1 can be adapted to the adaptive sampling of light fields from procedural shaders. In addition to computing bounds for the shader over a large parameter domain that we then adaptively refine, we now also compute bounds over a continuum of camera positions. For example, we can start with a large bounding box specifying all possible camera positions, and then adaptively refine it. Or, in the case of a two-plane parameterized light field, we could define the range of camera positions as a rectangular region on the camera plane.

It is not clear at this point how the acquired hierarchical light field can be used directly for rendering in interactive applications. However, a regularly sampled two-plane parameterized light field is easy to generate from the hierarchical one by interpolation. This approach does not resolve the relatively large memory requirements of light fields, but it should dramatically reduce the acquisition time.

5 Space Variant BRDFs

The situation gets even more complex when we also want to allow for changes in the illumination. The most reasonable approach for dealing with this situation seems to be storing a reflection model (BRDF) for every point on the object. That is, instead of precomputing the shader for *all* possible lighting situations (which would require even more space), we only determine the BRDF at every surface location (i.e. a *space-variant BRDF* by considering the effect of a single directional light source which can be pointing at the object from any direction.

As mentioned in the introduction, a space-variant BRDF is a six-dimensional function, and keeping a six-dimensional table is prohibitive in size. Therefore, a different representation has to be found. Again, we should be able to use AAto generate a relatively sparse, adaptive sampling of the shader, which is, however, not well suited for interactive rendering.

On the other hand, the graphics hardware is becoming more and more flexible, so that it is now possible to render certain simple reflection models where the parameters of the model can be varied across the surface [11]. This yields a limited form of space-variant BRDF, where the BRDF actually conforms to a single analytical reflection model, but its parameters can be texture-mapped and can therefore vary across the surface.

Unfortunately, the reflection models considered in [11] are not yet complex enough to capture all the effects that a procedural shader may produce. Other models that provide a general purpose basis for arbitrary effects do exist [12], but it is currently no possible to render them in hardware with space-variant parameters.

Once we have found a reflection model that is expressive enough for our purposes and can be rendered in hardware, we still have to determine its parameters in every point of the object from the hierarchical samples acquired with the adaptive sampling approach. This, again, is an open research problem.

6 Conclusion

In this section we have raised some issues regarding the sampling of complex procedural shaders as a preprocessing step for interactive rendering. We were able to describe a hierarchical sampling scheme that adaptively determines an appropriate sampling resolution for different parts of the shader. The application of this method to determining view-dependent information from a shader in such a way that it is efficient to use in interactive applications is an open problem. We were able to identify some issues arising with this subject, and hinted towards some possible solutions, but more research will have to be done for a complete solution.

References

[1] E. H. Adelson and J. R. Bergen. *Computational Models of Visual Processing*, chapter 1 (The Plenoptic Function and the Elements of Early Vision). MIT Press, Cambridge, MA, 1991.

[2] Alias/Wavefront. *OpenAlias Manual*, 1996.

[3] Elliot W. Cheney. *Introduction to Approximation Theory*. International series in pure and applied mathematics. McGraw-Hill, 1966.

[4] João L. D. Comba and Jorge Stolfi. Affine arithmetic and its applications to computer graphics. In *Anais do VII Sibgrapi*, pages 9–18, 1993.

[5] Luiz Henrique Figueiredo. Surface intersection using affine arithmetic. In *Graphics Interface '96*, pages 168–175, 1996.

[6] Luiz Henrique Figueiredo and Jorge Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. *Computer Graphics Forum*, 15(5):287–296, 1996.

[7] Steven J. Gortler, Radek Grzeszczuk, Richard Szelinski, and Michael F. Cohen. The Lumigraph. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 43–54, August 1996.

[8] Xianfeng Gu, Steven J. Gortler, and Michael F. Cohen. Polyhedral geometry and the two-plane parameterization. In *Rendering Techniques '97 (Proceedings of Eurographics Rendering Workshop)*, pages 1–12, June 1997.

[9] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, pages 289–298, August 1990.

[10] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics*, pages 158–176, 1998.

[11] Jan Kautz and Hans-Peter Seidel. Towards Interactive Bump Mapping with Anisotropic Shift-Variant BRDFs. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000*, pages 51–58, August 2000.

[12] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-linear approximation of reflectance functions. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 117–126, August 1997.

[13] Marc Levoy and Pat Hanrahan. Light field rendering. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 31–42, August 1996.

[14] Gavin Miller, Steven Rubin, and Dulce Ponceleon. Lazy decompression of surface light fields for precomputed global illumination. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)*, pages 281–292, March 1998.

[15] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-speed rendering using image composition. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 231–240, July 1992.

[16] Ramon E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, New Jersey, 1966.

[17] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. The synthesis and rendering of eroded fractal terrains. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pages 41–50, July 1989.

[18] Pixar. *The RenderMan Interface*. Pixar, San Rafael, CA, Sep 1989.

[19] Peter-Pike Sloan, Michael F. Cohen, and Steven J. Gortler. Time critical Lumigraph rendering. In *Symposium on Interactive 3D Graphics*, 1997.

[20] John M. Snyder. *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design Using Interval Analysis*. Academic Press, 1992.

[21] John M. Snyder. Interval analysis for computer graphics. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 121–130, July 1992.

[22] Steve Upstill. *The RenderMan Companion*. Addison Wesley, 1990.

[23] D. Wood, D. Azuma, K. Aldinger, B. Curless, T. Duchamp, D. Salesin, and W. Stuetzle. Surface Light Fields for 3D Photography. In *Proceedings of SIGGRAPH 2000*, pages 287–296, July 2000.

Chapter 10

Multi-Pass RenderMan

Marc Olano

Multi-Pass RenderMan

Marc Olano
SGI

1. The RenderMan Shading Language

RenderMan is an interface for talking to Renderers [Upstill90]. There are several RenderMan-compatible software renderers, the most well known of which is Pixar's PhotoRealistic RenderMan. One of the most interesting features of the RenderMan Interface is its shading language [Hanrahan90]. The RenderMan shading language is not the only shading language for software rendering (see Chapter 2), but it is often used as a standard of comparison thanks to its power and its popularity.

So, if RenderMan is "the standard", why aren't there any real-time RenderMan implementations? It isn't that we wouldn't want one. First, graphics hardware simply isn't capable enough yet. Second, the running time for a RenderMan shader can be arbitrarily long. Finally, real-time rendering encourages a different style of shader writing than software rendering.

1.1. Necessary Hardware Features

Why isn't interactive graphics hardware capable of RenderMan, real-time or not? The principle feature lacking on current graphics hardware is floating point. Machines with floating-point have been designed (e.g. PixelFlow), but none have made it to commercial availability. All existing graphics hardware store results and do most computations using clamped fixed point representations. In contrast, the RenderMan shading language has only one numeric representation - floating point. This provides great freedom to the shader writer. Quantities can be expressed in physical units; overflow and loss of precision are occasional problems, not something you worry about on every line of source; you can even have an enormous range in scale and precision across a single surface.

At the time [Peercy00] was written, most hardware also lacked the ability to look up texture results based on previous computations. Many current graphics systems support either pixel texture (interpret per-pixel framebuffer color as texture coordinates) or dependent texture (interpret previous texturing results as new texture coordinates). Either of these extensions is sufficient to allow RenderMan's indirect lookups.

If the necessary hardware capabilities were present, it would be completely possible to have a hardware accelerated RenderMan. We have demonstrated this using a software implementation of OpenGL (a modified version of the SGI OpenGL sample implementation). This modified OpenGL uses floating point for all computations and storage, it supports pixel texture and color matrix OpenGL extensions, and base OpenGL 1.2. This demo system translates any RenderMan shader into multiple OpenGL rendering passes.

1.2. Arbitrary running time

The RenderMan shading language is similar in structure to C. Like C, it is easy to write programs

that never stop. A shader that never finishes isn't terribly interesting on any rendering system, software or real-time. However, shaders can (and sometimes are!) thousands of lines long or loop for thousands of iterations.

More concretely, even if we had the ability to run RenderMan on graphics hardware we couldn't guarantee that **all** RenderMan shaders would run in real-time. Some

1.3. Programming style

The final factor preventing real-time RenderMan is shader programming style. It isn't that real-time shaders can't be written in RenderMan, but even with all the necessary hardware features, even perfectly ordinary RenderMan shaders may not run in real-time. Shaders written for real-time rendering typically rely heavily on stuffing as much computation as possible into the values stored in each texture, leaving the shader to combine these textures in interesting ways. Shaders written for RenderMan do use texture, but they can also use arbitrary computation. That's part of what makes the shading language so powerful.

2. Sample RenderMan shader

An example is worth a thousand words. This simple RenderMan shader creates a simple beach ball. The ball appearance is entirely procedurally generated, there no textures are used. This example demonstrates both how, given the right extensions, translating a RenderMan shader to multi-pass OpenGL can by quite straightforward, and how this doesn't necessarily give real-time RenderMan.

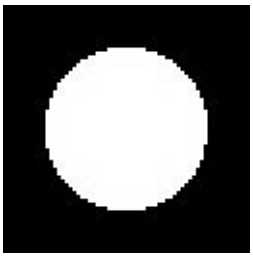
```
surface
beachball(
    uniform float Ka = 1, Kd = 1;
    uniform float Ks = .5, roughness = .1;
    uniform color starcolor = color (1,.5,0);
    uniform color bandcolor = color (1,.2,.2);
    uniform float rmin = .15, rmax = .4;
    uniform float npoints = 5;
     )
{
```

```
    color Ct;
    float angle, r, a, in_out;
    uniform float starangle = 2*PI/npoints;
    uniform point p0 = rmax*point(cos(0),sin(0),0);
    uniform point p1 = rmin*
        point(cos(starangle/2),sin(starangle/2),0);
    uniform vector d0 = p1 - p0;
    vector d1;

    angle = 2*PI * s;
    r = .5-abs(t-.5);
    a = mod(angle, starangle)/starangle;

    if (a >= 0.5)
        a = 1 - a;
    d1 = r*(cos(a), sin(a),0) - p0;
    in_out = step(0, zcomp(d0^d1));
    Ct = mix(mix(Cs, starcolor, in_out), bandcolor, step(rmax,r));

    /* specular shading model */
    normal Nf = normalize(faceforward(N,I));
    Oi = Os;
    Ci = Os * (Ct * (Ka * ambient() +
                     Kd * diffuse(Nf)) +
               Ks * specular(Nf,-normalize(I),roughness));
}
```

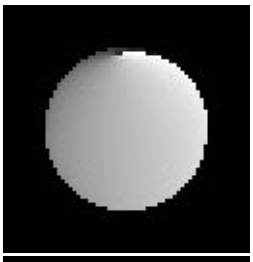
3. Passes for varying computation

Each line of text below is a pass used as part of the varying computation in the above shader. Corresponding images appear to the left, though no image appears for any pass that does not change the framebuffer. No optimizations are included in this example as they can make the correspondence between source code and passes harder to follow. In the thumbnails here, all values are clamped to [0,1] **for display purposes only**; the value stored in the floating point framebuffer or floating point textures still remain unclamped.

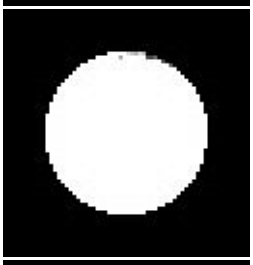
```
// set stencil for masking in later passes
```



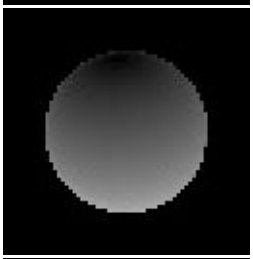
```
// draw geometry with 's' as color
angle = 2*PI * s;
```



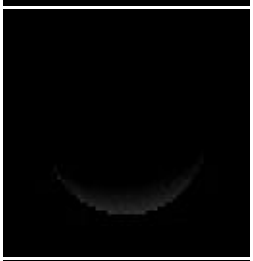
```
// use blend to multiply by 2*PI
angle = 2*PI * s;
// store in texture named "angle"
angle = 2*PI * s;
```



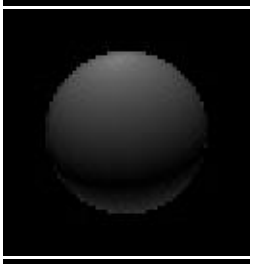
```
// draw geometry with 't' as color
r = .5-abs(t-.5);
```



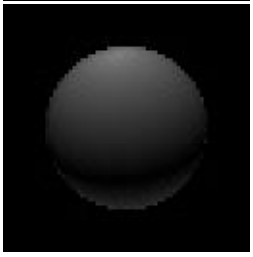
```
// use blend to subtract .5
r = .5-abs(t-.5);
```



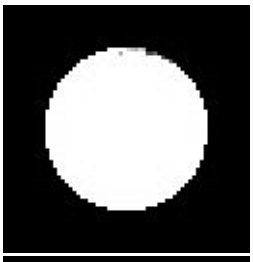
```
// copy through "abs" color table
r = .5-abs(t-.5);
```



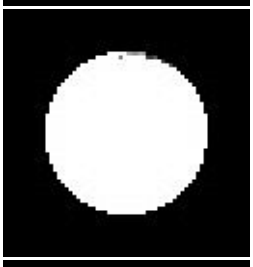
```
// blend: subtract from .5
r = .5-abs(t-.5);
// store in texture named "r"
r = .5-abs(t-.5);
```



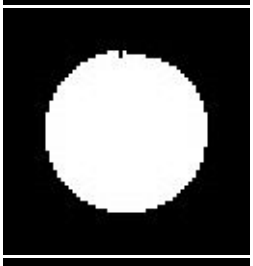
```
// load "angle" from texture
a = mod(angle, starangle)/starangle;
```



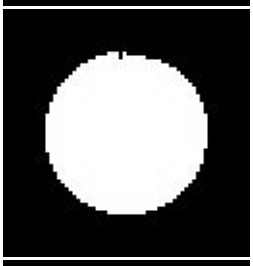
```
// blend: multiply by 1/starangle
a = mod(angle, starangle)/starangle;
```



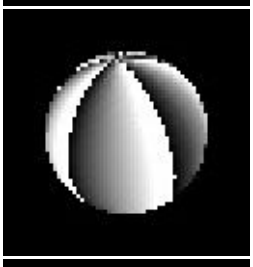
```
// copy through "floor" color table
a = mod(angle, starangle)/starangle;
```



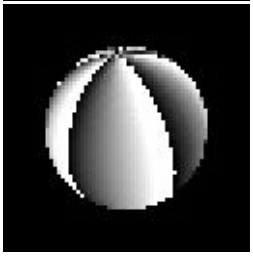
```
// blend: multiply by starangle
a = mod(angle, starangle)/starangle;
```



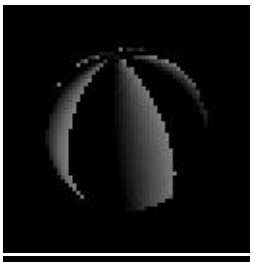
```
// blend: subtract from "angle"
a = mod(angle, starangle)/starangle;
```



```
// blend: multiply by 1/starangle
a = mod(angle, starangle)/starangle;
// store in texture named "a"
a = mod(angle, starangle)/starangle;
// load "a" from texture
if (a >= 0.5)
```

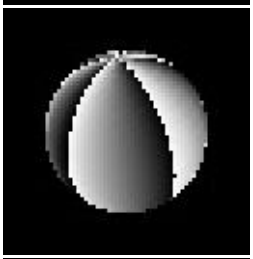


```
// blend: subtract .5
if (a >= 0.5)
```

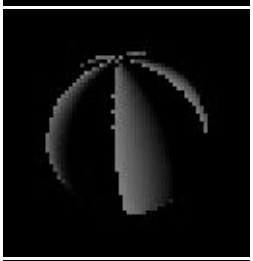


```
// alpha test: set stencil mask
if (a >= 0.5)
```

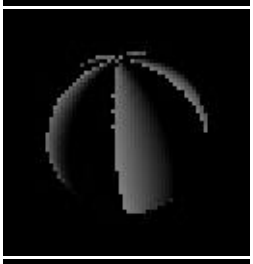
```
// load "a" from texture
    a = 1 - a;
```



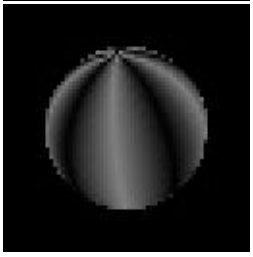
```
// blend: subtract from 1
    a = 1 - a;
```



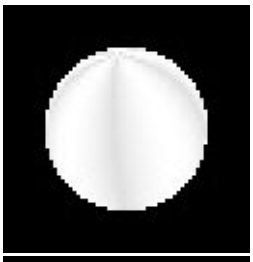
```
// load "a" & combine with stencil
    a = 1 - a;
// store in texture named "a"
    a = 1 - a;
```



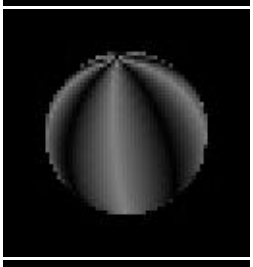
```
// load "a" from texture
d1 = r*(cos(a), sin(a),0) - p0;
```



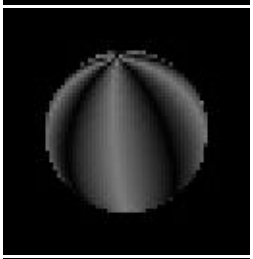
```
// copy through "cos" color table
d1 = r*(cos(a), sin(a),0) - p0;
// store in texture named "ftemp0"
```



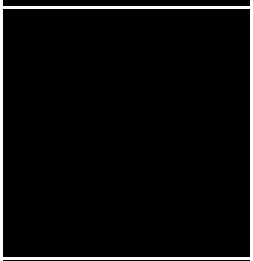
```
// load "a" from texture
d1 = r*(cos(a), sin(a),0) - p0;
```



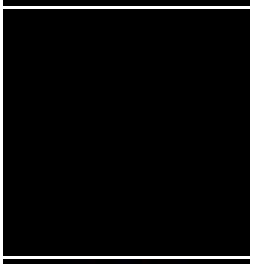
```
// copy through "cos" color table
d1 = r*(cos(a), sin(a),0) - p0;
// store in texture named "ftemp1"
```



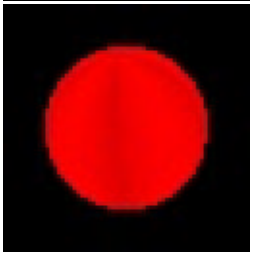
```
// load constant value of 0
d1 = r*(cos(a), sin(a),0) - p0;
```



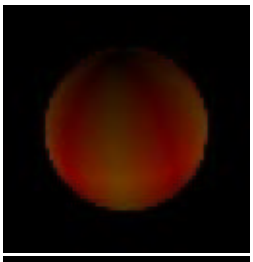
```
// load "ftemp0" into red
d1 = r*(cos(a), sin(a),0) - p0;
```



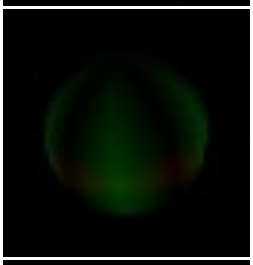
```
// load "ftemp1" into green
d1 = r*(cos(a), sin(a),0) - p0;
```



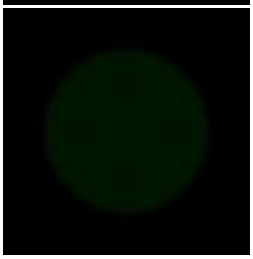
```
// blend: multiply by texture "r"
d1 = r*(cos(a), sin(a),0) - p0;
```



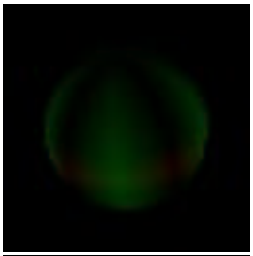
```
// blend: subtract uniform p0
d1 = r*(cos(a), sin(a),0) - p0
// store in texture named "d1"
d1 = r*(cos(a), sin(a),0) - p0
```



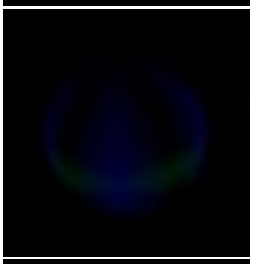
```
// load uniform d0
in_out = step(0, zcomp(d0^d1));
// color matrix: store in yzx order in "ctemp0"

// color matrix: store in zxy order in "ctemp1"
```



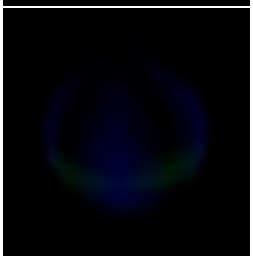
```
// load "d1" from texture
in_out = step(0, zcomp(d0^d1));
// color matrix: store in yzx order in "ctemp2"
```



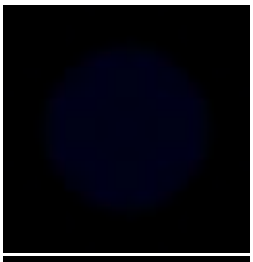
```
// color matrix: shuffle to zxy order
in_out = step(0, zcomp(d0^d1));
```



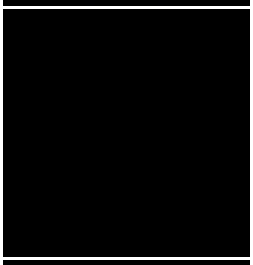
```
// blend: multiply by "ctemp0"
in_out = step(0, zcomp(d0^d1));
// store back into "ctemp0"
```



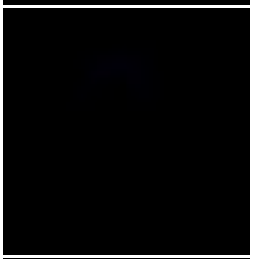
```
// load "ctemp1" from texture
in_out = step(0, zcomp(d0^d1));
```



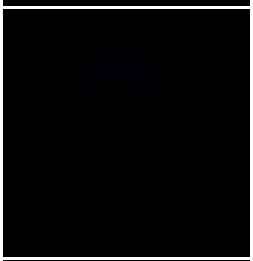
```
// blend: multiply by "ctemp2"
in_out = step(0, zcomp(d0^d1));
```



```
// blend: subtract "ctemp0"
in_out = step(0, zcomp(d0^d1));
```



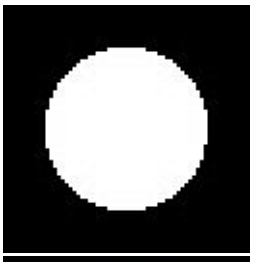
```
// blend: subtract "ctemp0"
in_out = step(0, zcomp(d0^d1));
```



```
// color matrix: copy z to all channels
in_out = step(0, zcomp(d0^d1));
// blend: subtract 0 (to shift step)
in_out = step(0, zcomp(d0^d1));
```

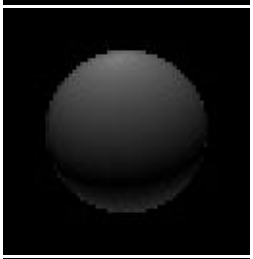
```
// copy through "step" color table
in_out = step(0, zcomp(d0^d1));
// store in texture named "in_out"
in_out = step(0, zcomp(d0^d1));
```

```
// load uniform Cs
...mix(Cs, starcolor, in_out)...
// load "in_out" into alpha
...mix(Cs, starcolor, in_out)...
```

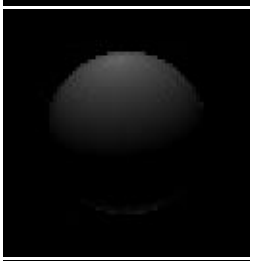


```
// blend: mix Cs and starcolor
...mix(Cs, starcolor, in_out)...
// store in texture named "ctemp0"
```

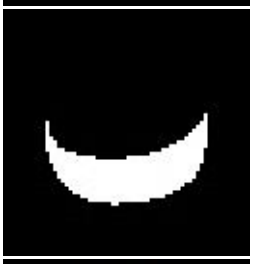
```
// load "r" from texture
...mix(..., bandcolor, step(rmax,r));
```



```
// blend: subtract from rmax
...mix(..., bandcolor, step(rmax,r));
```



```
// copy through "step" color table
...mix(..., bandcolor, step(rmax,r));
// store in texture named "ftemp0"
```

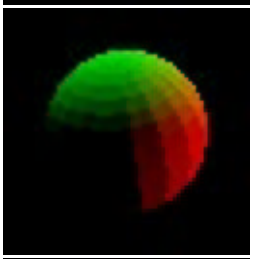


```
// load "ctemp0"
...mix(..., bandcolor, step(rmax,r));
// load "ftemp0" into alpha
...mix(..., bandcolor, step(rmax,r));
```

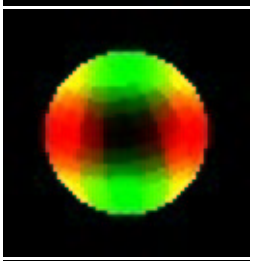
```
// blend: mix with bandcolor
...mix(..., bandcolor, step(rmax,r))
// store in texture named "Ct"
Ct = mix(...);
```

```
// draw geometry, with 'I' as color
...normalize(faceforward(N,I));
// store in texture named "ctemp0"
```

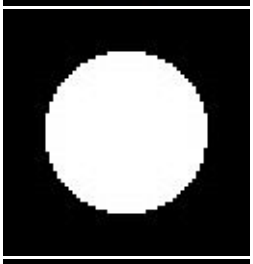
```
// draw geometry, with 'Ng' as color
...normalize(faceforward(N,I));
```



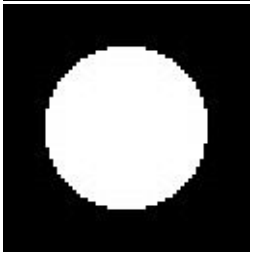
```
// blend: multiply by texture "ctemp0"
...normalize(faceforward(N,I));
```



```
// color matrix: add x+y+z
...normalize(faceforward(N,I));
```

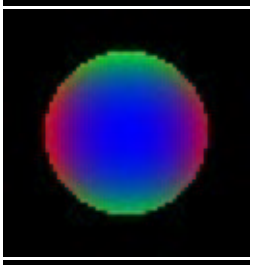


```
// copy through "flip" color table
...normalize(faceforward(N,I));
```

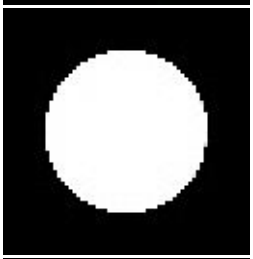


```
// blend: draw 'N' & multiply
...normalize(faceforward(N,I));
// store in texture named "ctemp0"
```

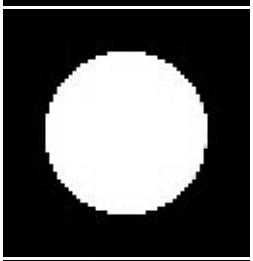
```
// blend: multiply (to square)
normal Nf = normalize(...);
```



```
// color matrix: sum channels
normal Nf = normalize(...);
```

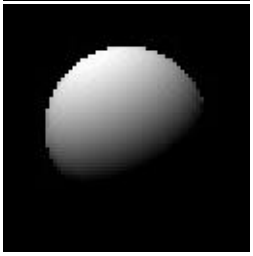


```
// copy through "invsqrt" color table
normal Nf = normalize(...);
```

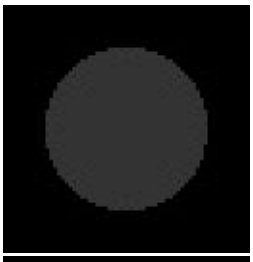


```
// blend: multiply by texture "ctemp0"
normal Nf = normalize(...);
// store in texture named "Nf"
normal Nf = normalize(...);
```

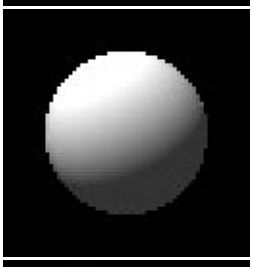
```
// Lighting passes omitted
Ci = ...(... + Kd * diffuse(Nf))...
// store in texture named "ctemp0"
```



```
// Lighting passes omitted
Ci = ...(Ka * ambient() + ...)...
```

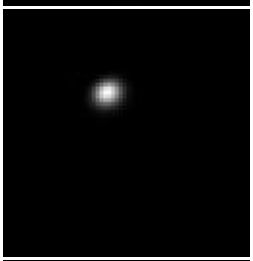


```
// blend: add "ctemp0"
Ci = ...(Ka * ambient() + ...)...
```

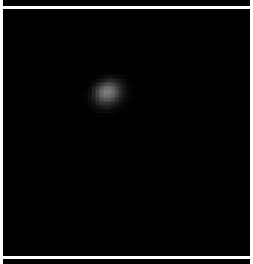


```
// blend: multiply by "Ct"
Ci = ...Ct * (...)...
// store in texture named "ctemp0"
```

```
// Lighting passes omitted
Ci = ...Ks * specular(...));
```



```
// blend: multiply by uniform 'Ks'
Ci = ...Ks * specular(...));
```



```
// blend: add "ctemp0" (diffuse & ambient)
Ci = Os * (...);
// blend: multiply by Os
Ci = Os * (...);
// load "Ci" outside object using stencil
Ci = Os * (...);
// store combined Ci into texture named "Ci"
Ci = Os * (...);
```

Chapter 11

Analysis of Shading Pipelines

John C. Hart

A Framework for Analyzing Real-Time Advanced Shading Techniques

John C. Hart
University of Illinois
jch@cs.uiuc.edu

Peter K. Doenges
Evans & Sutherland Computer Corp.
pdoenges@es.com

Abstract

Numerous techniques have been developed to perform advanced shading operations in real time. The real-time versions vary greatly from their original implementations due to the constraints of existing hardware and programming libraries. This paper introduces a grammar for articulating these modern real-time shading pipelines. We survey the existing techniques, using the grammar to classifies them into a taxonomy illustrating the commutations that differentiate the pipelines from the classical version and each other. The taxonomy follows a natural development that concludes with texture shading, which is applied to four advanced shaders to explore its versatility.

1 Introduction

The task of presenting a three-dimensional object on a two-dimensional display relies largely on perceptual cues the brain has evolved for resolving the three-dimensional spatial configuration of a scene from its projection onto the eye's two-dimensional retina. One of these cues is shading: the variation in color and intensity of a surface that indicates its position and orientation within a scene.

Consumer computer graphics has finally outgrown the classic lighting model composed of Lambertian diffuse and Phong/Blinn specular reflection that dominated hardware implementation for the past two decades. [Cook & Torrance, 1982] noted that the standard technique of matching the diffuse component with the surface material color and the specular component with the color of the light source was a good model for plastic, which consists of a suspension of diffusing pigments covered by a thin clear specular membrane. With the support of advanced shaders, consumer computer graphics will finally overcome its cliché plastic appearance.

Procedural shaders generate the color of a point on the surface by running a short program. The Renderman system, which contains a little language specifically developed for procedural shaders, is the current industry standard for procedural shading. Hanrahan suggests that Renderman, while adequate for production-quality shading, may not be the most appropriate choice for specifying real-time shaders [Hanrahan, 1999].

Shaders determine the color of light exiting a surface as a function of the light entering a surface and the properties of the surface. Shaders combine the cues of *lighting*, which determines how light uniformly interacts with the surface, and *texturing*, which determines how light nonuniformly interacts with the surface. We use the stationarity of the phenomena as a key differentiator between surface lighting and surface texturing. Hence, "texturing" is a feature parameterized in part by position whereas "lighting" is not.

Section 3 surveys current real-time advanced shading strategies. Finding information on these topics is not easy. The techniques have resulted as much from developers as from researchers, and these techniques appear in tutorials, how-to's, product specifications and reports more often than in journals and conference proceedings. This survey collects these ideas together in one place and presents the techniques in a uniform and organized manner to allow better comparison and understanding of their benefits and drawbacks.

This paper is in part a response to the keynote talk of the Eurographics/SIGGRAPH 1999 Graphics Hardware Workshop [Hanrahan, 1999]. This talk lamented the fact that there are many directions hardware vendors are considering to support advanced shading. This situation was best described by the slide in Figure 1. Section 3 develops a natural progression from the standard graphics pipeline through fragment lighting, multitexturing and multipass eventually concluding with texture shading.

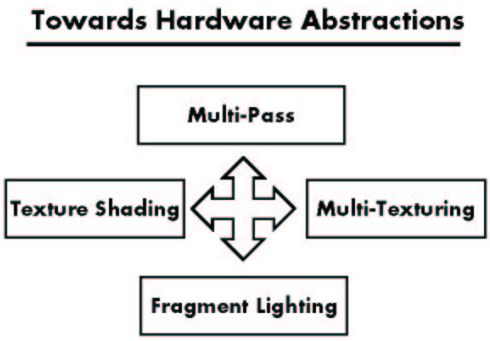


Figure 1. Slide 22 of [Hanrahan, 1999]

Hanrahan recommended that graphics hardware community should investigate solutions to this problem by "commuting the graphics pipeline." The grammar introduced in Section 2 provides a representation where such commutations can be articulated. analyzed and classified. Such grammars are already familiar in the analysis of rendering. A grammar associated with the rendering equation [Kajiya, 1985] has been used to classify the transport of light from its source to

the viewer as a sequence of symbols corresponding to emittance, occlusion, and specular and diffuse surface reflections.

This paper also follows up on the ideas of [McCool & Heidrich, 1999], which proposed a texture shader built on a multitexturing stack machine and a wider variety of texture coordinate generation modes. Section 4 begins to look in detail at what kinds of shaders and variations are possible using these advanced programmable shader techniques.

2 A Graphics Pipeline Grammar

This section develops a grammatical representation for the graphics shading pipeline. The symbols used in this grammar are listed in Figure 2.

$\mathbf{x}$	vertex in model coordinates
$\mathbf{u}$	surface parameterization (u,v)
$\mathbf{s}$	shading parameter vector $(N,V,R,H,\ldots)$
π	graphics pipeline from model to viewport coordinates
$\mathbf{x}_s$	pixel in viewport coordinates (x_s,y_s)
δ	rasterization (interpolation and sampling)
$\mathbf{c}$	color vector (R,G,B)
$\oplus$	color combination function
C	frame buffer
T	texture map
$\leftarrow$	assignment

Figure 2: Operators in the shading pipeline grammar.

We denote a two-dimensional surface parameterization as $\mathbf{u} = (u,v)$. We denote the shading parameters as a vector $\mathbf{s}$ that contains light source and surface material constants, as well as the local coordinate frame and the view vector. We use the vector $\mathbf{x}$ to represent a triangle vertex with its position in model coordinates, and $\mathbf{x}_s$ to denote the same point in viewport (screen) coordinates. The 2-D surface texture coordinates are an attribute of the vertex and are denoted $\mathbf{u}\,\mathbf{x}$. Likewise, the shading parameter vector is also a vertex attribute and is denoted $\mathbf{s}\,\mathbf{x}$. Note that since these functions accept a single parameter, we eliminate the use of paranthesis in favor of a grammatical expression.

We denote color $\mathbf{c} = (R,G,B)$. The map $\mathbf{p}$: $\mathbf{s} \rightarrow \mathbf{c}$ denotes a shader, a procedure that maps shading parameters to a color $\mathbf{c}$. The operator T: $\mathbf{u} \rightarrow \mathbf{c}$ is a 2-D texture map that returns a color $\mathbf{c}$ given surface texture coordinates $\mathbf{u}$.

We use capital letters to denote maps that are implemented with a lookup table, such as the texture map T. We will use the $\leftarrow$ operator to denote assignment to this table. For example, the framebuffer C:$\mathbf{x}_s \rightarrow \mathbf{c}$ is a mapping from screen coordinates $\mathbf{x}_s$ to a color $\mathbf{c}$. The frame buffer is implemented as a table, and assignment of an element $\mathbf{c}$ into this table at index $\mathbf{x}_s$ is denoted as $C\ \mathbf{x}_s \leftarrow \mathbf{c}$.

Most of the standard graphics pipeline can be decomposed into a general projection π that maps vertices from 3-D model coordinates $\mathbf{x}$ to 2-D screen coordinates $\mathbf{x}_s$, and a rasterization that takes these screen coordinate vertices and fills in the pixels of the polygon they describe using linear interpolation. It will be useful for the analysis of the aliasing artifacts to know exactly when attributes are interpolated across a polygon, as this signals when continuous functions are discretely sampled. We will indicate that a continuous function has been discretely sampled by rasterization with a delta function operator δ.

Hence, $\mathbf{x}$ is a polygon vertex in model-coordinates, $\pi\,\mathbf{x}$ is the screen coordinate corresponding to that point and $\delta\,\pi\,\mathbf{x}$ reminds us that the coordinates of that pixel were discretely interpolated from the screen coordinates of the polygon's vertices. The goal of the next section will be to articulate and analyze various techniques for assigning a value to the screen pixel $C\,\delta\,\pi\,\mathbf{x}$.

3 Procedural Shading Techniques

This section analyzes various graphics shading pipelines, including the standard pipeline, deferred shading, multipass, multitexturing, environment map techniques and texture shading. It also makes explicit shading techniques supported by these pipelines, including Phong mapping and environment mapped bump mapping.

3.1 Standard Graphics Pipeline Shading

The standard implementation of the modern graphics pipeline is dominated by the linear interpolation of vertex attributes at the end of the pipeline.

Gouraud Interpolation. The standard graphics pipeline implementation of lighting is expressed in this notation as

$$C\,\delta\,\pi\,\mathbf{x} \leftarrow \delta\,\mathbf{p}\,\mathbf{s}\,\mathbf{x}. \tag{1}$$

Lighting is computed per-vertex, and the resulting color is interpolated (using a technique known as Gouraud shading) across the pixels of the screen-space polygon by the rasterization engine.

Texture Mapping. Texturing is performed in screen coordinates and texture coordinates are assigned per-vertex and interpolated across the pixels of the screen-space polygon by the rasterization engine. Interpolated texture coordinates are then index into a texture map to determine a per-pixel texture color

$$C\,\delta\,\pi\,\mathbf{x} \leftarrow T\,\delta\,\mathbf{u}\,\mathbf{x}. \tag{2}$$

The aliasing artifacts introduced by texture mapping occur when the sampling rate of the delta function on the LHS of (2) (the resolution of the polygon's screen projection) disagrees with the sampling rate of the delta function on the RHS (the texture's resolution). Methods for resampling the texture map based on the MIP map [Williams, 1983] or the summed-area table [Crow, 1984] fix this problem by adjusting the sampling density on the RHS of (2) to match that of the LHS.

An additional though subtle issue with the δ function on the RHS of (2) is perspective correction. Since the projection π on the LHS performs a perspective divide, then the δ rasterization function on the RHS must also perform a per-pixel perspective divide.

Modulation. The results of lighting and texture mapping are combined using a modulation operator

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow (\delta\ \mathbf{p}\ \mathbf{s}\ \mathbf{x}) \oplus (T\ \delta\ \mathbf{u}\ \mathbf{x}). \quad (3)$$

In other words, the color of each pixel in the polygon's projection $\pi(\mathbf{x})$ is given by a blending operation of the pixel in the texture map *T* and the interpolated shading of the polygon's vertices.

3.2 Fragment Lighting

Fragment lighting is perhaps the most obvious way to implement lighting. It simply extends the per-vertex lighting currently present in graphics libraries to per-pixel computation. Fragment lighting thus computes the shading of each pixel as it is rasterized

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow \mathbf{p}\ \delta\ \mathbf{s}\ \mathbf{x}. \quad (4)$$

The shader parameters stored at each vertex are interpolated across the pixels of the polygon's projection, and a procedure is called for each pixel to synthesize the color of that pixel. Methods for Phong shading in hardware [Bishop & Weimer, 1986],[Peercy *et al.*, 1997] are based on fragment lighting, as are a variety of procedural shaders, both production [Hanrahan & Lawson, 1990] and realtime [Hart *et al.*, 1999].

Note that fragment lighting (4), which supports Phong shading interpolation, is a permutation of (1), which supports Gouraud shading interpolation. The juxtaposition of sampling δ and shader evaluation **p** suffices to change the representation from interpolating shader results (color) with shader parameters (e.g. surface normals).

Fragment lighting applies the entire procedure to each pixel before moving to the next. The main drawbacks to this technique is that interpolated vectors, such as the surface normal, need to be renormalized, which requires an expensive per-pixel square root operation. If this renormalization is approximated or ignored, the resulting artifacts can be incorrect shading, and this error increases with the curvature of the surface the polygon approximates.

The second drawback is the amount of per-pixel computation versus the amount of per-pixel time. Assuming a megapixel display and a 500 MIPS computer sustaining a 10 Hz frame rate limits procedures to 50 instructions. While a high-level VLIW instruction set could implement most shaders in 50 instructions, this would be a wasteful investment of resources since most shaders remain static, and the shader processor continue to repeatedly synthesize the same texture albeit for different pixels as the object moves across the screen.

The sampling rate of the δ in the LHS of (4) (the resolution of the polygon's projection) matches the sampling rate of the RHS (the resolution of the polygon sampling the shader). Hence aliasing occurs when this rate insufficiently samples the shader **p**. With the exception of the obvious and expensive supersampling technique, procedural shaders can be antialiased by bandlimiting [Norton, *et al.*, 1982] and a gradient magnitude technique [Rhoades, *et al.*, 1992], [Hart *et al.*, 1999], both which modify the texture procedure **p** to only generate signals properly sampled by the coordinates discretized by the δ.

3.3 Deferred Shading

Deferred shading [Molnar, 1992] implements procedural shading in two phases. In the first phase

$$T\ \delta\ \pi\ \mathbf{x} \leftarrow \delta\ \mathbf{s}\ \mathbf{x} \quad (5)$$

such that the shading parameters are stored in a fat texture map *T* which is the same resolution as the display, called the fat framebuffer. Once all of the polygons have been scan converted, the second phase makes a single shading pass through every pixel in the frame buffer

$$C\ \mathbf{x}_s \leftarrow \mathbf{p}\ T\ \mathbf{x}_s \quad (6)$$

replacing the color with the results of the procedure applied to the stored solid texture coordinates. In this matter, the application of **p**, the shader, is deferred until all of the polygons have been projected, such that the shader is only applied to visible sections of the polygons.

Deferred shading applies all of the operations of the shader expression to a pixel before the next pixel is visited, and so suffers the same process limitations as fragment lighting. Unlike fragment lighting, deferred shading has a fixed number of pixels to visit, which provides a constant execution speed regardless of scene complexity.

In fact, the main benefit of deferred shading is the reduction of its shading depth complexity to one. This means a shader is evaluated only once for each pixel, regardless of the number of overlapping objects that project to that pixel. Since some shaders can be quite complex, only applying them to visible pixel can save a significant amount of rendering time.

The main drawbacks for deferred shading is the size of the fat framebuffer *T*. The fat framebuffer contains every variable used by the shader, including surface normals, reflection vectors, and the location/direction and color of each light source.

One possible offset to the large frame buffer is to generate the frame in smaller chunks, trading space for time. It is not yet clear whether the time savings due to deferred shading's unit depth complexity makes up for the multiple renderings necessary for this kind of implementation.

Antialiasing is another drawback of deferred shading since the procedural texture is generated in a separate step of the algorithm than the step where the samples have been recorded from the δ. Deferred shading thus precludes the use of efficient polygon rasterization antialiasing methods such as coverage masks. Unless a significant amount of auxiliary information is also recorded, previous procedural texturing antialiasing algorithms do not easily apply to deferred shading.

However, with the multi-sample antialiasing found in many recent graphics controllers, supersampling appears to be the antialiasing technique of choice, and is certainly the most directly and generally scalable antialiasing solution across all segments of the graphics market. While the deferred shading frame buffer would have to be as large as the sampling space, this still seems to be a feasible and attractive direction for further pursuit.

Since all of the information needed by shader is held in the fat frame buffer per pixel, the channels of the framebuffer would need to be signed and generally of higher precision than the resulting color to prevent error accumulation in complex shaders.

3.4 Environment Map Lighting

There are a variety of texture coordinate modes that implement useful features. Recall that an environment map is an image of the incoming luminance of a point in space.

Spheremap. Environment mapping is most commonly supported in hardware by the spheremap mode. This texture coordinate generation mode assigns a special projection of the reflection vector *R* component of the shading information **s** to the texture coordinate **u** of the vertex **x**

$$\mathbf{u}\,\mathbf{x} \leftarrow \pi^{R}\,\mathrm{R}\,\mathbf{s}\,\mathbf{x}. \tag{7}$$

This projection requires a per-vertex square root that is handled during texture coordinate generation. The texture map consists of a mirror-ball image of the surrounding environment, which is combined with standard lighting by (3). This notation reveals how environment mapping avoids the interpolation of normalized vectors, by instead interpolating and sampling the projection of the reflection vector as a texture coordinate. This inexact approximation can create artifacts on large polygons spanning high-curvature areas.

Phong/Gloss Mapping. One problem with vertex lighting is that since vertex colors are interpolated, specular highlights that peak inside polygon faces do not get properly sampled. Specular highlights can be simulated using (13) to generate an environment spheremap consisting of a black sphere with specular highlights. This allows current graphics API's to support Phong highlights on polygon faces, using (7) for texture coordinate generation and (3) for modulating texturing with lighting. These highlights could be considered the incoming light from a diffused area light source, thereby softening the appearance of real-time shaded surfaces.

One significant advantage of the environment map is that it can contain the incoming light information from any number of light sources. Without it, we have a shading complexity that is linear in the number of light sources, often requiring a separate pass/texture for each light source. Putting all of the light source information into an environment map reduces the complexity to constant in the number of light sources.

Environment Mapped Bump Mapping. Phong mapping can also be used to approximate bump mapping

$$\mathrm{C}\,\delta\,\pi\,\mathbf{x} \leftarrow \mathrm{T}\,\delta\,((\mathbf{u}\,\mathbf{x}) + (\mathrm{T'}\,\delta\,\mathbf{u'}\,\mathbf{x})) \tag{8}$$

where T' is a bump map texture that, instead of a color, returns a 2-D vector (the partial derivatives of the bump map height field) that offsets the index into the environment map. Comparing EMBM (8) to standard texture mapping (2) shows precisely where the perturbation occurs. This form of bump mapping is supported by Direct3D 6.0, but requires hardware support for the offsetting of texture indices by a texture result. It would be interesting to investigate what other effects are possible by offsetting a texture index with the result of another texture.

3.5 Multipass Rendering

Modern graphics API's have limited resources that are often exhausted implementing a single effect. Multipass algorithms render the scene several times, combining a new effect with the existing result. Each pass can include an additional shader element, including lighting effects and texture layers. Each pass follows the form

$$\mathrm{C}\,\delta\,\pi\,\mathbf{x} \leftarrow (\mathrm{C}\,\delta\,\pi\,\mathbf{x}) \oplus ((\delta\,\mathbf{p}\,\mathbf{s}^{*}\,\mathbf{x}) \oplus (\mathrm{T}^{*}\,\delta\,\mathbf{u}^{*}\,\mathbf{x})) \tag{9}$$

where the asterisk indicates operators that typically change for each pass. The image composition operators of OpenGL 1.2.1 support a variety of frame combination operators.

Multipass is a SIMD approach that distributes procedural operations across the screen, applying a single operation to every screen pixel before moving to the next operation in the shading expression.

The benefit of multipass rendering is its flexibity. Any number of passes and combinations can be supported, and can be used to support full-featured shading languages [Proudfoot, 1999], [Olano, *et al.*, 2000].

The drawback of multipass rendering is its execution time. Each pass typically requires the re-rasterization of all of the geometry. Furthermore, storage and combination of frame buffer images can be incredibly slow. In many OpenGL implementations, it is faster to display a one-polygon-per-pixel mesh texture mapped with an image than to try to write the image directly to the screen.

Multipass rendering benefits from a retained (scene graph) mode since the input object data rarely changes from pass to pass. If multipass is used from a static viewpoint, then the polygons need only be rasterized once, and each pass can be performed on screen space vertices ($\pi\,\mathbf{x}$) instead of model space vertices **x**, specifically

$$\mathrm{C}\,\delta\,\mathbf{x}_{\mathrm{s}} \leftarrow (\mathrm{C}\,\delta\,\mathbf{x}_{\mathrm{s}}) \oplus ((\delta\,\mathbf{p}\,\mathbf{s}^{*}\,\mathbf{x}) \oplus (\mathrm{T}^{*}\,\delta\,\mathbf{u}^{*}\,\mathbf{x})) \tag{10}$$

Such a system would combine the benefits of deferred shading (compare (5) and (6)) with multipass since the shading would be deferred until after polygon projection. The shader would be executed only on the polygons that survived clipping, but this includes more than just the visible polygons.

As each new frame is composed with a previous frame, a low-pass filter should remove high frequencies from the new frame before composition. It is not yet clear what interference patterns could be created when composing two images with energy at nearly the same

frequency. Some situations could cause a beating pattern at a lower, more noticeable frequency.

As the results of the shading expression are accumulated in the frame buffer, the precision of the frame buffer needs to be increased beyond the final color output precision. To best accomodate a variety of individual shading operations, the intermediate frame buffers need to support signed values.

Accumulation Buffer. Perhaps the most obvious multipass technique is the accumulation buffer

$$C\ \delta\ \pi^*\ \mathbf{x} \leftarrow (C\ \delta\ \pi^*\ \mathbf{x}) \oplus (\delta\ \mathbf{p}\ \mathbf{s}\ x) \tag{11}$$

where π^* indicates that the projection is perturbed. This perturbation supports antialiasing, motion blur, depth of field and, when combined with a shadowing technique, soft shadows.

Shadow Mask. The multipass method for rendering shadows via the shadow mask [Williams, 1978] is given by the following steps

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow \delta\ \mathbf{p}\ \mathbf{s}\ \mathbf{x},$$

$$C'\ \delta\ \pi\ \mathbf{x} \leftarrow \delta\ \mathbf{p}\ \mathbf{s}'\ \mathbf{x},$$

$$C^l\ \delta\ \pi^l\ \mathbf{x} \leftarrow \delta\ \mathbf{x},$$

$$\alpha\ C\ \mathbf{x}_s \leftarrow (z\ C\ \mathbf{x}_s) > (z\ C^l\ \pi^l\ \pi^{-1}\ \mathbf{x}_s),$$

$$C\ \mathbf{x}_s \leftarrow (\alpha\ C\ \mathbf{x}_s)*(C\ \mathbf{x}_s) + (1 - \alpha\ C\ \mathbf{x}_s)*(C'\ \mathbf{x}_s).$$

where **s** contains ambient lighting parameters and **s**' contains diffuse and specular. The superscript *l* indicates a frame buffer C^l and projection π^l for the light source. The expression $\alpha\ C\ \mathbf{x}_s$ returns the alpha channel of the frame buffer C at position $\mathbf{x}_s$, and likewise the z operator returns the depth channel. Analyzing the shadow mask algorithm in this notation reveals several opportunities for special hardware to support parallel operation and pass combination.

Shadow Volumes. Multipass techniques usually rely heavily on the stencil buffer to either restrict the shader's operations to a section of the screen, or to store a temporary result of a shading operation. For example, the shadow volume method can be expressed

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow \delta\ \mathbf{p}\ \mathbf{s}\ \mathbf{x},$$

$$s\ C\ \delta\ \pi\ \mathbf{x} \leftarrow (s\ C\ \delta\ \pi\ \mathbf{x})\ \text{OR}\ ((z\ \delta\ \pi\ \mathbf{x}') > (z\ C\ \delta\ \pi\ \mathbf{x}')),$$

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow (s\ C\ \delta\ \pi\ \mathbf{x})\ ?\ (\delta\ \mathbf{p}\ \mathbf{s}'\ \mathbf{x}).$$

In this example, the object vertices are denoted **x** and the shadow volume vertices are **x**'. The operator s(C) returns the stencil buffer value from the frame buffer C. The vector **s** contains ambient shading parameters whereas **s**' contains diffuse and specular parameters.

3.6 Multitexturing

Multitexturing allows different textures to be combined on a surface. Multitexturing is a SIMD approach that distributes procedural operations across data, performing a single operation on the entire texture before moving to the next operation.

OpenGL 1.2.1 supports chained multitexturing

$$C\delta\pi\mathbf{x} \leftarrow T'''\delta\mathbf{u}'''\mathbf{x} \oplus (T''\delta\mathbf{u}''\mathbf{x} \oplus (T'\delta\mathbf{u}'\mathbf{x} \oplus T\delta\mathbf{u}\mathbf{x})). \tag{12}$$

where the $\oplus$ symbol denotes one of the OpenGL texture modes, either decal, modulate or blend. Direct3D appears to be extending these modes to allow a larger variety of texture expressions.

Multitexturing avoids the antialiasing roadblocks encountered by deferred shading because multitexturing defers the shading to the texture map, then projects the result onto the screen. This sets up the opportunity for shading aliases, which are more tolerable, without affecting rasterization aliases, which are more distracting.

Antialiasing in a multitexturing system could be accomplished by antialiasing each of the component textures. MIP mapping of multitexture components is one method used to filter the texture components.

Since the textures are used as components to shading equations, higher precision texture maps are needed to accumulate intermediate results, especially if scales greater than one are allowed. Signed texture values are also necessary.

3.7 Texture Shading

Texture shading stores shading information in the texture coordinates and maps. In its simplest form, it is expressed as

$$T\ \delta\ \mathbf{u} \leftarrow \mathbf{p}\ \mathbf{s}\ \delta\ \mathbf{u} \tag{13}$$

where the texture coordinate vector **u** indexes local illumination and texturing information **s**, and **p** applies a shader to this information, storing the resulting color in the texture map. The texture map is then applied to the surface using (2), which now takes responsibility for both texturing and lighting [Kautz & McCool, 1999] . Such techniques require special texture generation modes such that the texture coordinates contain a portion of the shader expression. These methods are demonstrated in Section 4.

Fat Texture Map. Texture shading occurs on a surface, which is parameterized by a two-dimensional coordinate system. A fat texture map could be considered that stores a vector of shading parameter instead of simply colors

$$C\ \delta\ \pi\ \mathbf{x} \leftarrow \mathbf{p}\ T\ \delta\ \mathbf{u}\ \mathbf{x}. \tag{14}$$

The parameters stored in the fat texture map might include vectors such as surface normals and tangents, or cosines such as the dot product of the surface normal and the light direction. This model of texture

shading is similar to deferred shading, replacing the fat frame buffer with a fat texture map.

Incorporating texture shading into multitexturing replaces the fat texture map with a collection of standard-sized texture maps each containing a sub-expression result of a complex shader expression. McCool proposed a multitexturing algebra based on a stack machine, allowing more complex texture expressions. McCool's proposal for dot products overlooks the sines of the angles between vectors, which could be useful for rendering hair.

It is interesting that the linear interpolation across the polygon interpolates the indices across the parameter vectors stored in texture memory. This allows the interpolation of normals and other shading parameters to be precomputed, such that only the index **u** need be interpolated [Kilgard, 1999].

Solid Mapping. Texture shading was used to perform solid texturing in OpenGL without any extensions [Carr, et al., 2000]. The technique assumed that the mapping $\mathbf{u}$: $\mathbf{x} \rightarrow \mathbf{u}$ is one-to-one (such that images of the object's polygons do not overlap in the texture map *T*). The object's polygons are rasterized into the texture map

$$T\ \delta\ \mathbf{u}\ \mathbf{x} \leftarrow \delta\ \mathbf{s}\ x, \tag{15}$$

where the shading parameters, in this case the solid texture coordinates (s,t,r), are stored as a color $\{R = s\ ;\ G = t\ ;\ B = r\}$ in the texture map *T*. A second pass

$$T\ \delta\ \mathbf{u} \leftarrow \mathbf{p}\ T\ \delta\ \mathbf{u} \tag{16}$$

replaces the texture map contents (s,t,r) with a color (R,G,B) generated by the procedural shader **p** on the solid texture coordinates. The texture map now contains a procedural solid texture that can be mapped back onto the object using standard texture mapping (2).

4 Applications

In the previous section, we followed a natural progression of techniques to support the real-time implementation of advanced shading models. This progression concluded with texture shading, which, when supported by multitexturing and multipass rendering, provides a powerful tool for implementing advanced shaders, though the full power of this tool is not yet completely understood. We explore the capabilities of texture shading by considering the implementation of a variety of advanced shaders.

These advanced shaders require more information than the standard surface normal and reflection vector currently available. This information can be encoded as dot products, as recommended by [McCool & Heidrich, 1999]. The coordinates and vectors used by these shaders are enumerated in Figure 3.

u	the point on the surface whose illumination properties we are interested in;
N	the unit *surface normal* perpendicular to the tangent plane of the surface at **u**;
T	principal *tangent vector* used to fix the orientation of the coordinate frame at u for anisotropic shading;
L	a light-dependent unit *light vector* anchored at u in the direction of one of possibly many light sources;
V	the view-dependent unit *view vector* anchored at u in the direction of the viewer;
R	the light-dependent unit *light reflection vector* equal to $2(N \cdot L)N - L$;
H	the light- and view-dependent unit *halfway vector* equal to $L+V$ normalized (constant for orthographic projection and directional light sources);

Figure 3: Shading parameters.

One method for implementing advanced shaders is to precompute its results for all possible inputs. We consider the equivalence classes of the reflectance function of a surface $\rho(u,v,\theta_i,\phi_i,\theta_r,\phi_r)$ where u,v denotes a point on the surface, θ_i,ϕ_i are the elevation and azimuth of a light vector L on this point, and θ_r,ϕ_r are the elevation and azimuth of the viewing direction V. (We use the term BRDF although many shaders are not actually bidirectional [Lewis, 1994]). We will denote equivalence classes by replacing parameters of the plenoptic function with the symbol $\cdot$, as shown in Figure 4.

$\rho\ (\cdot,\cdot,\theta_i,\phi_i,\theta_r,\phi_r)$	BRDF
$\rho\ (\cdot,\cdot,\theta_i,\cdot,\cdot,\cdot)$	Diffuse, e.g. Lambert's law
$\rho\ (\cdot,\cdot,\theta_i,\cdot,\theta_r,\cdot)$	Isotropic, e.g. $N \cdot L$, $N \cdot V$
$\rho\ (\cdot,\cdot,\theta_i,\phi_i+\cdot,\theta_r,\phi_r+\cdot)$	Specular, e.g. $N \cdot L$, $N \cdot V$, $V \cdot R$
$\rho\ (\cdot,\cdot,\theta_i,\phi_i,\cdot,\cdot)$	Anisotropic diffuse, e.g. $N \cdot L$, $T \cdot L$
$\rho\ (u,v,\cdot,\cdot,\cdot,\cdot)$	Texturing
$\rho\ (u,v,\theta_i,\cdot,\cdot,\cdot)$	Diffuse bump mapping
$\rho\ (u,v,\theta_i,\phi_i+\cdot,\theta_r,\phi_r+\cdot)$	Specular bump mapping

Figure 4: Equivalence classes of reflectance functions.

We investigate the various advanced texturing and shading techniques within these equivalence classes and use the classes to determine if precomputation and storage is feasible within the implementation technique.

[Cabral *et al*., 1999] showed how a general BRDF could be applied through the environment spheremap by assigning to it the reflected luminance instead of the incident luminance. While a technique for interpolating these luminance maps was described, this technique relies on a large number of environment maps discretized over the possible view positions.

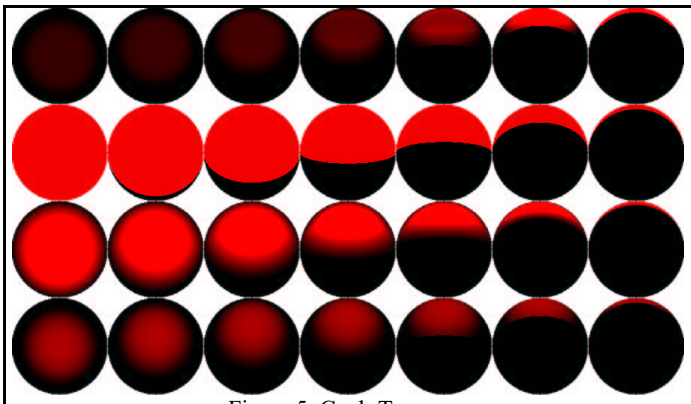


Figure 5: Cook-Torrance.

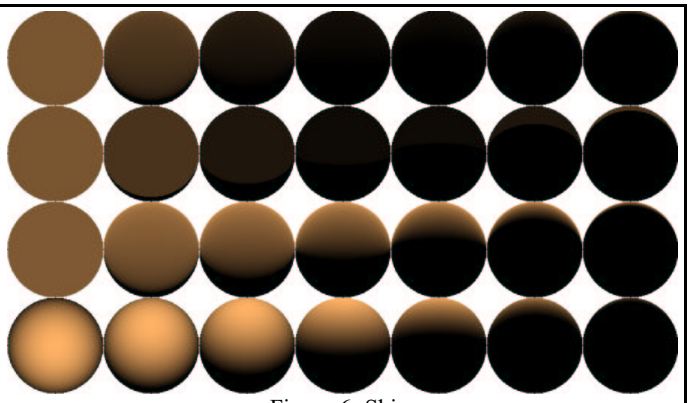


Figure 6: Skin.

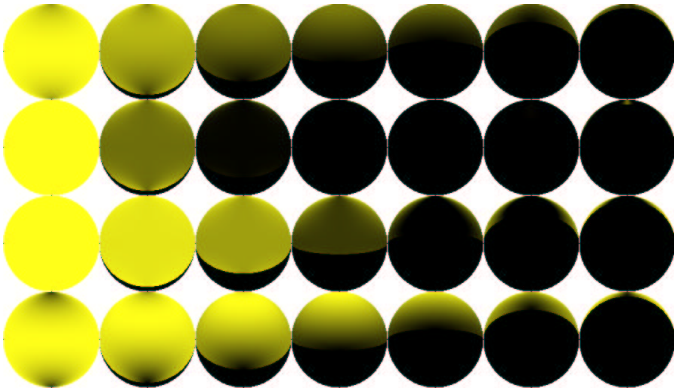


Figure 7: Hair.

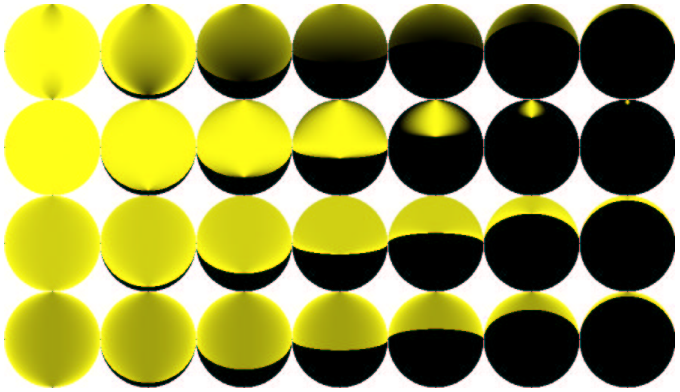


Figure 8: Fake Fur.

Companies such as nVidia have announced interest and support in 3-D texture maps, they are not currently available in an efficient form through current graphics API's. A 3-D texture map would be capable of storing reflectance information for specular reflectance and even diffuse bump mapping.

The advanced shaders we investigated typically use at least four distinct values as their parameters, which precludes the use of a texture map to lookup precomputed results. However, these advanced shaders are created from separable 2-D reflectance functions that can be combined to form the final multidimensional shader. [Kautz & McCool, 1999] decomposed 4-D BRDF's into a sequence of separable functions of 2-D reflectance functions. Basing the separability of shaders on the model instead of a general decomposition has the added benefit of supporting parameterization of the model, requiring recomputation of only the component whose parameter has changed, or even the real-time control of the blending operations between the individual lookup textures.

4.1 Cook-Torrance

The Torrance-Sparrow local illumination model is a highly empirical physically based method that is both experimentally and physically justified. The most common implementation of the Torrance-Sparrow model is the Cook-Torrance approximation [Cook & Torrance, 1982] of the specular component

$$\rho = \frac{FDG}{\pi (N \cdot V)(N \cdot L)}. \tag{17}$$

The Fresnel effect is the total reflection of light glancing off of a surface at an angle shallower than the critical angle, which is modeled as

$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1+\frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right) \tag{18}$$

where $c = V \cdot H$ and $g^2 = \eta^2 + c^2 - 1$. Computed directly, the divisions and square root would be costly, though feasible, for direct hardware implementation of this term. Alternatively, an approximation or a two-dimensional lookup table indexed by g and c would also suffice. The constants for the Frenel term vary with wavelength, so separate F terms can be computed for each color channel, resulting in a highlight that changes hue with intensity. The Frenel effect is plotted in the second row of Figure 5.

The roughness term is a distribution of the orientation of the microfacets, which is typically modeled by the Beckmann distribution function,

$$D = \frac{1}{4m^2(N \cdot H)^4}\exp\left(\frac{(N \cdot H)^2 - 1}{m^2(N \cdot H)^2}\right), \tag{19}$$

parameterized by the surface roughness m. The Beckmann distribution function for m=0.6 is plotted in the fourth row of Figure 5. This could be implemented with a 2-D texture map parameterized by $N \cdot H$ and m, which would also allow the roughness to vary across a surface.

The geometric attenuation factor G accounts for self-shadowing

$$G = \min\left\{1, \frac{2(N \cdot H)}{V \cdot H}(N \cdot V), \frac{2(N \cdot H)}{V \cdot H}(N \cdot L)\right\} \tag{20}$$

as the smaller of one, the inverse of the percentage of blocked incident light, and the inverse of the percentage of blocked reflected light, and is demonstrated in the third row of Figure 5. The geometry term consists of four cosines $N \cdot H$, $V \cdot H$, $N \cdot V$ and $N \cdot L$. However, the implementation can be separated into the product of two texture maps. A base 2-D texture map of $2(N \cdot H)/(V \cdot H)$, modulated by a 1-D texture maps containing either $N \cdot V$ or $N \cdot L$. (If the API supports scaling by the texture coordinate, these 1-D texture maps could be eliminated.)

Note that the full Cook-Torrance implementation, shown in the first row of Figure 5, requires four cosines $N \cdot H$, $N \cdot L$, $V \cdot H$, and $V \cdot L$. Precomputation and storage of the lighting model would result in a four-dimensional table equivalent to the BRDF. Hence, programmable shading remains a more efficient implementation for this lighting model.

4.2 Multilayer Shaders

Multilayer shaders decompose reflected light into a surface scattered component and a sub-surface scattered component at each layer. There many applications of multilayer shaders, including materials such as skin, leaves, tree canopies and shallow ponds.

Whereas Lambertian reflection is constructed from geometric principles, Seeliger's model [Hanrahan & Krueger, 1993] is constructed from first principles in physics as

$$\rho = \frac{N \cdot L}{(N \cdot L) + (N \cdot V)} \tag{21}$$

It scatters light more uniformly than Lambert's law, providing a softer appearance similar to skin. Compare the fourth row (Lambertian) with the third row (Seeliger) of Figure 6. This lighting model is isotropic (but not bidirectional). It could be precomputed using a two-dimensional texture map indexed by the cosines $N \cdot L$ and $N \cdot V$, or even by arithmetic on two texture coordinates.

The Henyey-Greenstein function was used to model the scattering of light by particles in a given layer

$$p(L \cdot V) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 - 2gL \cdot V)^{3/2}} \tag{22}$$

which is parameterized by the mean cosine of the direction of scattering g. This scattering function is plotted as intensity in the second row of Figure 6.

The scattering function was used as a probability distribution function for the Monte Carlo model that constructed a full BRDF by sampling a hemisphere of incoming light and measuring the exiting light on the same hemisphere. However, the Henyey-Greenstein function could also be used as an opacity function for texture layers, as demonstrated in the first row of Figure 6. As such, it can be implemented as a 2-D texture indexed by the cosine $L \cdot V$ and the scattering parameter g. One possible improvement is to implement the Henyey-Greenstein scattering using the EMBM enhancement (8).

Alternatively, the entire skin reflection function could be implemented as a 3-D specular BRDF, indexed by $N \cdot L$, $N \cdot V$ and $L \cdot V$.

4.3 Anisotropic Shaders

Anisotropic lighting models require a grain tangent direction in the reflectance coordinate frame, and must also account for self-shadowing. The most common use for anisotropic reflection is in the simulation of hair and fur, but can also be used for brushed metals and grassy fields.

A BRDF for hair was modeled [Kajiya & Kay, 1989] with a diffuse component given by the sine of the angle between the hair direction and the light vector

$$\rho_d = \sin(T, L) = \sqrt{1 - (T \cdot L)^2} \tag{23}$$

and the specular component as the sum of the products of the sines and cosines of the angle between the hair direction and the light vector and the view vector, raised to a specular exponent

$$\rho_s = \left((T \cdot L)(T \cdot V) + \sqrt{1 - (T \cdot L)^2}\sqrt{1 - (T \cdot V)^2}\right)^n. \tag{24}$$

Figure 7 shows these shading models. The fourth row is diffuse. The third row is specular with exponent one and the second row is specular with exponent 8. Note that the tangent dot products may be negative, such that raising to an even power changes the sign. The diffuse and specular components are combined in the first row.

The diffuse reflection can be implemented with as a 1-D texture map, indexed by $T \cdot L$. (The cosine-to-sine conversion is so fundamental that perhaps it bears hardware implementation.) The specular reflection function can be implemented as a 2-D texture map, indexed by $T \cdot L$ and $T \cdot V$. Alternatively, for directional light and orthographic views, this can be implemented using the tangent vector T as the texture coordinate, and using a texture transformation matrix whose first row is L and second row is V [Heidrich & Seidel, 1998].

This model was further enhanced for efficient use in the entertainment industry [Goldman, 1997]. The scattering of light by hair and fur is approximated by

$$p = \frac{(T \times L) \cdot (T \times V)}{\left|T \times L\right\|T \times V\right|} \tag{25}$$

the cosine of the dihedral angle between the hair-light plane and the hair-view plane. Compare the fourth row of Figure 8, which contains the diffuse and specular terms, with the third row, which plots the scattering function.

An opacity function for hair is given by an inverted Gaussian

$$\alpha(V) = 1 - \exp\left(\frac{-k\sqrt{1-(V\cdot T)^2}}{V\cdot N}\right) \quad (26)$$

where k is a constant equal to the projected area of each strand of hair times the number of hair strands, both per unit square. This opacity function is plotted (for k=0.1) as intensity in the second row of Figure 8. These terms are collected to form a general reflectance model for hair

$$\rho_{\text{hair}} = \alpha(V)(1 - s\alpha(L))\left(\frac{1+p}{2}k_r + \frac{1-p}{2}k_t\right)(k_d\rho_d + k_s\rho_s) \quad (27)$$

which combines constants of reflection k_r and transmission k_t (backlighting), and diffuse k_d and specular k_s reflection. The fraction s is used to control the degree of self-shadowing of hair. This expression can be implemented as a multipass rendering, or a multitexturing if the API supports the operations. This result is demonstrated in the first row of Figure 8.

4.4 Non-Photorealistic Shaders

While photorealism has been a longstanding goal of computer graphics, a significant amount of attention has also been paid to the use of graphics for illustration and visualization. The fundamental problem in non-photorealistic rendering is silhouette detection. The silhouette of an object occurs where the surface normal is perpendicular to the view vector, which could be indicated by the reflectance

$$\rho = 1-(1-V\cdot N)^n \quad (28)$$

where the exponent n indicates the crispness of the silhouette.

Shading in illustrations is often performed by hash marks, which often follow the tangent directions of the surface, and hardware shaders based on this form of shading would need the tangent vectors in addition to the surface normal to properly orient a prestored or synthesized hash texture. One could implement such hashing using a hashed spheremap.

5 Conclusion

We tackled the problem of analyzing present shader technology. We introduced a grammar capable of representing the fundamental nature of and differences between real-time shading techniques. We used this grammar to compare features of the standard pipeline with deferred rendering, multipass, multitexturing, texture shading and environment map techniques. We also evaluated these techniques with respect to a variety of advanced shaders.

We found that the natural progression of the real-time shader techniques leads to texture shading supported by multitexturing and multipass. We also found that storage of the BRDF is inefficient, and advanced shading procedures are too complex to implement directly, but they can however be assembled by multitexture components that consist of 2-D texture maps indexed by coordinates generated from dot products of shader vector variables.

5.1 Future Work

Analyzing real-time shading pipelines using the grammar provides a basis for innovation, and makes various commutations easier to consider. We expect this comparison may inspire new techniques based on innovative permutations of the parameterization, shader, projection and interpolation operations.

We also expect the grammar to grow more specific, providing a more detailed view of the specific channels and coordinates used for various shading effects.

Due to the constraints of time, we have omitted bump mapping, procedural texturing and the noise function from this discussion. Half of procedural shading is procedural texturing, though most of the attention on advanced shading has focused on lighting and local illumination models.

5.2 Acknowledgments

This work was supported in full by a consulting contract through the Evans & Sutherland Computer Corp. Conversations with Kurt Akeley, John Buchanan, Nate Carr, Rich Ehlers, Alain Fournier, Eric Haines, Pat Hanrahan, Chuck Hansen, Masaki Kameya, Michael McCool, Marc Olano and Steve Tibbitts were very useful in uncovering the details of many of these real-time shading techniques.

Bibliography

[Banks, 1994] D. C. Banks. Illumination in Diverse Codimensions. Computer Graphics (Proceedings of SIGGRAPH '94), 1994, pp. 327-334.

[Bishop & Weimer, 1986] Gary Bishop and David M. Weimer. Fast Phong Shading, Computer Graphics 20(4), (Proceedings of SIGGRAPH 86), Aug. 1986, pp. 103-106.

[Cabral *et al.*, 1999] Brian Cabral and Marc Olano and Philip Nemec. Reflection Space Image Based Rendering, Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, Aug. 1999, pp. 165-170.

[Carr, et al., 2000] Nate Carr, John Hart and Jerome Maillot. The Solid Map: Methods for Generating a 2-D Texture Map for Solid Texturing. Proc. Western Computer Graphics Symposium, Mar. 2000.

[Cook & Torrance, 1982] R. L. Cook and K. E. Torrance. A Reflectance Model for Computer Graphics, ACM Transactions on Graphics, 1 (1), January 1982, pp. 7-24.

[Crow, 1984] Franklin C. Crow. Summed-area Tables for Texture Mapping, Computer Graphics 18(3), (Proceedings of SIGGRAPH 84), July 1984, pp. 207-212.

[Goldman, 1997] Dan B. Goldman. Fake Fur Rendering, Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, Aug. 1997, pp. 127-134.

[Hanrahan & Lawson, 1990] Pat Hanrahan and Jim Lawson. A Language for Shading and Lighting Calculations, Computer Graphics 24(4), (Proceedings of SIGGRAPH 90), Aug. 1990, pp. 289-298.

[Hanrahan & Krueger, 1993] Pat Hanrahan and Wolfgang Krueger. Reflection from Layered Surfaces Due to Subsurface Scattering, Proceedings of SIGGRAPH 93, Aug. 1993, pp. 165-174.

[Hanrahan, 1999] Patrick Hanrahan. Real Time Shading Languages. Keynote, Eurographics/SIGGRAPH Workshop on Graphics Hardware, Aug. 1999

[Hart *et al.*, 1999] John C. Hart, Nate Carr, Masaki Kameya, Stephen A. Tibbitts and Terrance J. Coleman. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation, 1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware, Aug., 1999, pp. 45-53.

[Heckbert, 1990] Paul S. Heckbert. Adaptive Radiosity Textures for Bidirectional Ray Tracing, Computer Graphics (Proceedings of SIGGRAPH 90), 24 (4), August 1990, pp. 145-154

[Heidrich & Seidel, 1998] W. Heidrich and H.-P. Seidel. Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware. Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP) 1998.

[Kajiya & Kay, 1989] James T. Kajiya and Timothy L. Kay. Rendering Fur with Three Dimensional Textures, Computer Graphics (Proceedings of SIGGRAPH 89), *23 (3)*, July 1989, pp. 271-280.

[Kajiya, 1985] James T. Kajiya. Anisotropic Reflection Models, Computer Graphics (Proceedings of SIGGRAPH 85), *19 (3)*, July 1985, pp. 15-21.

[Kajiya, 1986] James T. Kajiya. The Rendering Equation, Computer Graphics (Proceedings of SIGGRAPH 86), *20(4)*, August 1986, pp. 143-150.

[Kautz & McCool, 1999] Jan Kautz and Michael D. McCool. Interactive Rendering with Arbitrary BRDFs using Separable Approximations, Eurographics Rendering Workshop 1999, June 1999.

[Kilgard, 1999] Mark J. Kilgard. A Practical and Robust Bump-mapping Technique for Today's GPUs. nVidia Technical Report. Feb. 2000.

[Lastra *et al.*, 1995] Anselmo Lastra and Steven Molnar and Marc Olano and Yulan Wang. Real-Time Programmable Shading, 1995 Symposium on Interactive 3D Graphics, April 1995, pp. 59-66.

[Lewis, 1994] R. R. Lewis. Making Shaders More Physically Plausible, Computer Graphics Forum, *13 (2)*, January 1994, pp. 109-120.

[Norton, *et al.*, 1982] Norton, Alan, Alyn P. Rockwood and Phillip T. Skolmoski. Clamping: A method for antialiased textured surfaces by bandwidth limiting in object space. *Computer Graphics 16*(3), (Proc. SIGGRAPH 82), July 1982, pp. 1-8.

[McCool & Heidrich, 1999] Michael D.McCool and Wolfgang Heidrich. Texture shaders, 1999 SIGGRAPH/Eurographics Workshop on Graphics Hardware, August 1999, pp. 117-126.

[Molnar, 1992] Steven Molnar and John Eyles and John Poulton. PixelFlow: High-speed rendering using image composition, Computer Graphics (Proceedings of SIGGRAPH 92), *26 (2)*, July 1992, pp. 231-240.

[Olano & Lastra, 1998] Marc Olano and Anselmo Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System, Proceedings of SIGGRAPH 98, Computer Graphics Proceedings, Annual Conference Series, July 1998, pp. 159-168.

[Olano, *et al.*, 2000] Marc Olano, et al., Interactive Multi-Pass Programmable Shading. To appear: Proc. SIGGRAPH 2000.

[Peercy *et al.*, 1997] Mark Peercy and John Airey and Brian Cabral. Efficient Bump Mapping Hardware, Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, Aug. 1997, pp. 303-306.

[Proudfoot, 1999] Kekoa Proudfoot. Real Time Shading Language Description, Version 4. Nov. 1999.

[Rhoades, *et al.*, 1992] Rhoades, John, Greg Turk, Andrew Bellm Andrei State, Ulrich Neumann and Amitabh Varshney. Real-Time Procedural Textures. Proc. Interactive 3-D Graphics Workshop, 1992. pp. 95-100.

[Stalling & Zöckler, 1997] D. Stalling and M. Zöckler and H.-C. Hege Fast Display of Illuminated Field Lines. IEEE Transactions on Visualization and Computer Graphics, 3(2), 1997, pp. 118-128.

[Williams, 1978] Lance Williams. Casting Curved Shadows on Curved Surfaces, Computer Graphics (Proceedings of SIGGRAPH 78), *12(3)*, Aug. 1978, pp. 270-274.

[Williams, 1983] Lance Williams. Pyramidal Parametrics, Computer Graphics (Proceedings of SIGGRAPH 83), *17 (3)*, July 1983, pp. 1-11.

Chapter 12

Bibliography

References

[1] ANDREWS, H., AND HUNT. *Digital Image Restoration*. The Johns Hopkins University Press, 1977.

[2] APODACA, A. A., AND GRITZ, L. *Advanced RenderMan: Creating CGI for motion pictures*. Morgan Kaufmann, 2000.

[3] ASHIKHMIN, M., PREMOZE, S., AND SHIRLEY, P. A Microfacet-Based BRDF Generator. In *Proc. ACM SIGGRAPH* (July 2000), pp. 65–74.

[4] ATI. *Pixel Shader Extension*, 2000. Specification document, available from http://www.ati.com/online/sdk.

[5] ATI. *Vertex Shader Extension*, 2001. Specification document, available from http://www.ati.com/online/sdk.

[6] BANKS, D. Illumination in Diverse Codimensions. In *Proc. SIGGRAPH* (July 1994), pp. 327–334.

[7] BASTOS, R., HOFF, K., WYNN, W., AND LASTRA, A. Increased Photorealism for Interactive Architectural Walkthroughs. *1999 ACM Symposium on Interactive 3D Graphics* (April 1999), 183–190.

[8] BERGERON, P. Shadow Volumes for Non-Planar Polygons. In *Proc. Graphics Interface* (May 1985), pp. 417–418. Extended abstract.

[9] BERGERON, P. A General Version of Crow's Shadow Volumes. *IEEE CG&A 6*, 9 (Sept. 1986), 17–28.

[10] BLINN, J. Me and my (fake) shadow. *IEEE CG&A 8*, 1 (Jan. 1988), 82–86.

[11] BLINN, J. F. Models of light reflection for computer synthesized pictures. In *Computer Graphics (SIGGRAPH '77 Proceedings)* (July 1977), pp. 192–198.

[12] BLINN, J. F. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (Aug. 1978), pp. 286–292.

[13] BLINN, J. F., AND NEWELL, M. E. Texture and reflection in computer generated images. *Communications of the ACM 19* (1976), 542–546.

[14] BLYTHE, D., GRANTHAM, B., AND KILGARD, M. J. Lighting and shading techniques for interactive applications. In *SIGGRAPH 1999 Course Notes* (Aug. 1999).

[15] BLYTHE, D., GRANTHAM, B., KILGARD, M. J., MCREYNOLDS, T., AND NELSON, S. R. Advanced graphics programming techniques using OpenGL. In *SIGGRAPH 1999 Course Notes* (Aug. 1999).

[16] BOLIN, M. R., AND MEYER, G. W. A Perceptually Based Adaptive Sampling Algorithm. In *Proc. ACM SIGGRAPH* (July 1998), pp. 299–310.

[17] BROTMAN, L., AND BADLER, N. Generating Soft Shadows with a Depth Buffer Algorithm. *IEEE CG&A 4*, 10 (Oct. 1984), 71–81.

[18] CABRAL, B., MAX, N., AND SPRINGMEYER, R. Bidirectional Reflection Functions From Surface Bump Maps. In *Proc. SIGGRAPH* (July 1987), pp. 273–281.

[19] CABRAL, B., OLANO, M., AND NEMEC, P. Reflection space image based rendering. In *Computer Graphics (SIGGRAPH '99 Proceedings)* (Aug. 1999), pp. 165–170.

[20] CHIN, N., AND FEINER, S. Near Real-Time Shadow Generation Using BSP Trees. In *Proc. SIGGRAPH* (Aug. 1989), vol. 23, pp. 99–106.

[21] CHRYSANTHOU, Y., AND SLATER, M. Shadow Volume BSP Trees for Computation of Shadows in Dynamic Scenes. In *SIGGRAPH Symp. on Interactive 3D Graphics* (Apr. 1995), pp. 45–50.

[22] COHEN, M., AND WALLACE, J. *Radiosity and Realistic Image Synthesis*. Academic Press, 1993.

[23] COOK, R. L. Shade Trees. In *Proc. SIGGRAPH* (July 1984), pp. 223–231.

[24] CROW, F. Shadow Algorithms for Computer Graphics. In *Proc. SIGGRAPH* (July 1977), vol. 11, pp. 242–248.

[25] DANA, K. J., GINNEKEN, B. V., NAYAR, S. K., AND KOENDERINK, J. J. *Columbia-Utrecht Reflectance and Texture Database*. `http://www.cs.columbia.edu/CAVE/curet/`, 1999.

[26] DEBEVEC, P. E. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Computer Graphics (SIGGRAPH '98 Proceedings)* (July 1998), pp. 189–198.

[27] DEBEVEC, P. E., AND MALIK, J. Recovering high dynamic range radiance maps from photographs. In *Computer Graphics (SIGGRAPH '97 Proceedings)* (Aug. 1997), pp. 369–378.

[28] DEYOUNG, J., AND FOURNIER, A. Properties of Tabulated Bidirectional Reflectance Distribution Functions. In *Proc. Graphics Interface* (May 1997), pp. 47–55.

[29] DIEFENBACH, P. *Pipeline Rendering: Interaction and Realism throught Hardware-Based Multi-pass Rendering*. PhD thesis, Department of Computer and Information Science, 1996.

[30] DIEFENBACH, P., AND BADLER, N. Pipeline Rendering: Interactive refractions, reflections and shadows. *Displays: Special Issue on Interactive Computer Graphics 15*, 3 (1994), 173–180.

[31] DIEFENBACH, P., AND BADLER, N. Multi-Pass Pipeline Rendering: Realism For Dynamic Environments . *1997 ACM Symposium on Interactive 3D Graphics* (April 1997), 59–70.

[32] DUFF, T. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. In *Proc. ACM SIGGRAPH* (July 1992), pp. 131–138.

[33] EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. *Texturing and Modeling*, second ed. Academic Press, 1998.

[34] ENGLER, D. R. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. In *Proc. ACM SIGPLAN* (1996), pp. 160–170.

[35] ERNST, I., RÜSSELER, H., SCHULZ, H., AND WITTIG, O. Gouraud bump mapping. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware* (1998), pp. 47–54.

[36] FOURNIER, A. Separating reflection functions for linear radiosity. In *Eurographics Rendering Workshop* (June 1995), pp. 383–392.

[37] FREEMAN, W., AND ADELSON, E. The Design and Use of Steerable Filters. *IEEE Transaction on Pattern Analysis and Machine Intelligence 13*, 9 (Sept. 1991), 891–906.

[38] FUCHS, H., GOLDFEATHER, J., HULTQUIST, J., SPACH, S., AUSTIN, J., BROOKS, JR., F., EYLES, J., AND POULTON, J. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. In *Proc. SIGGRAPH* (July 1985), vol. 19, pp. 111–120.

[39] GOLUB, G., AND VAN LOAN, C. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, Maryland, 1983.

[40] GONDEK, J., MEYER, G., AND NEWMAN, J. Wavelength Dependent Reflectance Functions. In *Proc. ACM SIGGRAPH* (July 1994), pp. 213–220.

[41] GOOCH, B., SLOAN, P.-P., GOOCH, A., SHIRLEY, P., AND RIESENFELD, R. Interactive technical illustration. In *ACM Symposium on Interactive 3D Graphics* (1999), pp. 31–38.

[42] GORTLER, S., GRZESZCZUK, R., SZELINSKI, R., AND COHEN, M. The Lumigraph. In *Proc. SIGGRAPH* (Aug. 1996), pp. 43–54.

[43] GREENE, N. Applications of World Projections. In *Proceedings of Graphics Interface '86* (May 1986), pp. 108–114.

[44] GRITZ, L., AND HAHN, J. BMRT: A Global Illumination Implementation of the RenderMan Standard. *Journal of Graphics Tools 1*, 3 (1996), 29–47.

[45] GUENTER, B., KNOBLOCK, T., AND RUF, E. Specializing shaders. In *Proc. SIGGRAPH* (Aug. 1995), pp. 343–350.

[46] HAEBERLI, P., AND AKELEY, K. The accumulation buffer: Hardware support for high-quality rendering. In *Proc. SIGGRAPH* (Aug. 1990), pp. 309–318.

[47] HAEBERLI, P., AND SEGAL, M. Texture mapping as a fundamental drawing primitive. In *Fourth Eurographics Workshop on Rendering* (June 1993), pp. 259–266.

[48] HALL, R. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, 1989.

[49] HANRAHAN, P. *Radiosity and Realistic Image Synthesis*. Academic Press, 1993, ch. Rendering Concepts.

[50] HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)* (Aug. 1990), pp. 289–298.

[51] HANSEN, P. Introducing pixel texture. In *Developer News*. Silicon Graphics Inc., May 1997, pp. 23–26.

[52] HE, X., TORRANCE, K., SILLION, F., AND GREENBERG, D. A comprehensive physical model for light reflection. In *Proc. SIGGRAPH* (July 1991), pp. 175–186.

[53] HEIDRICH, W. *High-quality Shading and Lighting for Hardware-accelerated Rendering*. PhD thesis, Universität Erlangen-Nürnberg, 1999.

[54] HEIDRICH, W., KAUTZ, J., SLUSALLEK, P., AND SEIDEL, H.-P. Canned lightsources. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)* (1998).

[55] HEIDRICH, W., AND SEIDEL, H. Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware. In *Image and Multi-dimensional DSP Workshop (IMDSP)* (1998).

[56] HEIDRICH, W., AND SEIDEL, H.-P. View-independent environment maps. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware* (1998), pp. 39–45.

[57] HEIDRICH, W., AND SEIDEL, H.-P. Realistic, hardware-accelerated shading and lighting. In *Computer Graphics (SIGGRAPH '99 Proceedings)* (Aug. 1999).

[58] HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. An image-based model for realistic lens systems in interactive computer graphics. In *Graphics Interface '97* (1997), pp. 68–75.

[59] HEIDRICH, W., WESTERMANN, R., SEIDEL, H.-P., AND ERTL, T. Applications of pixel textures in visualization and realistic image synthesis. In *ACM Symposium on Interactive 3D Graphics* (1999). Accepted for publication.

[60] INC., S. G. *Pixel Texture Extension*, Dec. 1996. Specification document, available from http://www.opengl.org.

[61] JENSEN, H., AND CHRISTENSEN, P. Efficient Simulation of Light Transport in Scenes with Participating Media using Photon Maps. In *Proc. ACM SIGGRAPH* (July 1998), pp. 311–320.

[62] KAJIYA, J. T. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (Aug. 1986), pp. 143–150.

[63] KAUTZ, J. Hardware Rendering with Bidirectional Reflectances. Tech. Rep. TR-99-02, Dept. Comp. Sci., U. of Waterloo, 1999.

[64] KAUTZ, J. Interactive Reflections with Arbitrary BRDFs. Tech. Rep. TR-99-XX, Dept. Comp. Sci., U. of Waterloo, 1999.

[65] KAUTZ, J., AND MCCOOL, M. Interactive Rendering with Arbitrary BRDFs using Separable Approximations. In *Tenth Eurographics Workshop on Rendering* (June 1999), pp. 281–292.

[66] KAUTZ, J., AND MCCOOL, M. D. Interactive Rendering with Arbitrary BRDFs using Separable Approximations. In *Eurographics Rendering Workshop* (June 1999).

[67] KAUTZ, J., AND MCCOOL, M. D. Approximation of Glossy Reflection with Prefiltered Environment Maps. In *Proc. Graphics Interface* (May 2000), pp. 119–126.

[68] KAUTZ, J., VÁZQUEZ, P.-P., HEIDRICH, W., AND SEIDEL, H.-P. Unified approach to prefiltered environment maps. In *submitted* (2000).

[69] KELLER, A. Instant Radiosity. In *Proc. SIGGRAPH* (Aug. 1997), pp. 49–56.

[70] KILGARD, M. *OpenGL-based Real-Time Shadows*. `http://reality.sgi.com/mjk_asd/tips/rts/`, 1997.

[71] KILGARD, M. J. Realizing OpenGL: Two implementations of one architecture. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware* (1997).

[72] KILGARD, M. J. A practical and robust bump-mapping technique for today's GPU's. Tech. rep., NVIDIA Corp., Feb. 2000. Available at http://www.nvidia.com/.

[73] KOENDERINK, J., VAN DOORN, A., AND STAVRIDI, M. Bidirectional Reflection Distribution Function Expressed in Terms of Surface Scattering Modes. In *European Conference on Computer Vision* (1996), pp. 28–39.

[74] LAFORTUNE, E., FOO, S.-C., TORRANCE, K., AND GREENBERG, D. Non-linear approximation of reflectance functions. In *Proc. SIGGRAPH* (Aug. 1997), pp. 117–126.

[75] LAFORTUNE, E., AND WILLEMS, Y. Using the modified Phong reflectance model for physically based rendering. Tech. Rep. CW197, Dept. Comp. Sci., K.U. Leuven, 1994.

[76] LARSON, G. W., RUSHMEIER, H., AND PIATKO, C. A Visibility Matching Tone Reproduction Operator for High Dyn amic Range Scenes. *IEEE Transactions on Visualization and Computer Graphics 3*, 4 (Oct.–Dec. 1997), 291–306.

[77] LASTRA, A., MOLNAR, S., OLANO, M., AND WANG, Y. Real-time programmable shading. *1995 Symposium on Interactive 3D Graphics* (April 1995), 59–66. ISBN 0-89791-736-7.

[78] LENGYEL, J. E. Real-Time Fur. In *Rendering Techniques '00 (Proc. Eurographics Workshop on Rendering)* (2000), Springer, pp. 243–256.

[79] LEVOY, M., AND HANRAHAN, P. Light field rendering. In *Proc. SIGGRAPH* (Aug. 1996), pp. 31–42.

[80] LEWIS, R. Making shaders more physically plausible. In *Eurographics Workshop on Rendering* (June 1993), pp. 47–62.

[81] LINDHOLM, E., KILGARD, M., AND MORETON, H. A User-Programmable Vertex Engine. In *Proc. ACM SIGGRAPH* (Aug. 2001).

[82] LISCHINSKI, D., AND RAPPOPORT, A. Image-Based Rendering for Non-Diffuse Synthetic Scenes. *Nineth Eurographics Workshop on Rendering* (June 1998), 301–314.

[83] LITWINOWICZ, P. Processing images and video for an impressionistic effect. In *Proc. SIGGRAPH* (Aug. 1997), pp. 407–414.

[84] LOKOVIC, T., AND VEACH, E. Deep Shadow Maps. In *Proc. ACM SIGGRAPH* (July 2000), pp. 385–392.

[85] LOZANO, R., ET AL. Colorimetry. Tech. Rep. 15.2, Commision internationale de l'éclairage (CIE), 1986.

[86] MCCOOL, M. Analytic Antialiasing With Prism Splines. In *Proc. SIGGRAPH* (Aug. 1995), pp. 429–436.

[87] MCCOOL, M. Shadow Volume Reconstruction. Tech. Rep. CS-98-06, University of Waterloo Department of Computer Science, 1998.

[88] MCCOOL, M. D. Shadow Volume Reconstruction from Depth Maps. *ACM Trans. on Graphics 19*, 1 (Jan. 2000), 1–26.

[89] MCCOOL, M. D., ANG, J., AND AHMAD, A. Homomorphic Factorization of BRDFs for High-Performance Rendering. In *Proc. ACM SIGGRAPH* (Aug. 2001).

[90] MCCOOL, M. D., AND HEIDRICH, W. Texture Shaders. In *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware* (1999), pp. 117–126.

[91] MCREYNOLDS, T., BLYTHE, D., GRANTHAM, B., AND NELSON, S. Advanced graphics programming techniques using OpenGL. In *SIGGRAPH 1998 Course Notes* (July 1998).

[92] MEYER, G. W. Wavelength Selection for Synthetic Image Generation. *CVGIP 41* (1988), 57–79.

[93] MILLER, G., AND HOFFMAN, R. Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments. In *SIGGRAPH '84 Course Notes – Advanced Computer Graphics Animation* (July 1984).

[94] MILLER, G. S. P., RUBIN, S., AND PONCELEON, D. Lazy Decompression of Surface Light Fields for Precomputed Global Illumination. *Nineth Eurographics Workshop on Rendering* (June 1998), 281–292.

[95] MINNAERT, M. The reciprocity principle in lunar photometry. *Astrophysical Journal 93* (May 1941), 403–410.

[96] MITCHELL, D. P. Robust ray intersection with interval arithmetic. In *Proc. Graphics Interface* (May 1990), pp. 68–74.

[97] MÖLLER, T., AND HAINES, E. *Real-Time Rendering*. A. K. Peters, 1999.

[98] MOLNAR, S., EYLES, J., AND POULTON, J. PixelFlow: High-speed rendering using image composition. In *Proc. SIGGRAPH* (July 1992), pp. 231–240.

[99] NAYAR, S. K. Catadioptric omnidirectional camera. In *IEEE Conference on Computer Vision and Pattern Recognition* (June 1997), pp. 482–488.

[100] NEUMANN, L., AND NEUMANN, A. Photosimulation: interreflection with arbitrary reflectance models and illuminations. *Computer Graphics Forum 8*, 1 (Mar. 1989), 21–34.

[101] OFEK, E., AND RAPPOPORT, A. Interactive reflections on curved objects. In *Proc. SIGGRAPH* (July 1998), pp. 333–342.

[102] OLANO, M. *A Programmable Pipeline for Graphics Hardware*. PhD thesis, University of North Carolina at Chapel Hill, 1999.

[103] OLANO, M., AND LASTRA, A. A shading language on graphics hardware: The pixelflow shading system. In *Proc. SIGGRAPH* (July 1998), pp. 159–168.

[104] OWENS, J. D., DALLY, W. J., KAPASI, U. J., RIXNER, S., MATTSON, P., AND MOWERY, B. Polygon Rendering on a Stream Architecture. In *Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware* (2000), pp. 23–32.

[105] PEACHEY, D. Solid texturing of complex surfaces. In *Proc. ACM SIGGRAPH* (July 1985), pp. 279–286.

[106] PEERCY, M. Linear Color Representations for Full Spectral Rendering. In *Proc. ACM SIGGRAPH* (Aug. 1993), pp. 191–198.

[107] PEERCY, M., AIREY, J., AND CABRAL, B. Efficient bump mapping hardware. In *Computer Graphics (SIGGRAPH '97 Proceedings)* (Aug. 1997), pp. 303–306.

[108] PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. Interactive multi-pass programmable shading. *Proceedings of SIGGRAPH 2000* (July 2000), 425–432. ISBN 1-58113-208-5.

[109] PERLIN, K. An image synthesizer. *Computer Graphics (Proceedings of SIGGRAPH 85) 19*, 3 (July 1985), 287–296.

[110] PHONG, B.-T. Illumination for computer generated pictures. *Comm. ACM 18*, 6 (June 1975), 311–317.

[111] POULIN, P., AND FOURNIER, A. A model for anisotropic reflection. In *Proc. SIGGRAPH* (Aug. 1990), pp. 273–282.

[112] PRESS, W., TEUKOLSKY, S., VETTERLING, W., AND FLANNERY, B. *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.)*. Cambridge University Press, 1992.

[113] PROUDFOOT, K., MARK, W. R., HANRAHAN, P., AND TZVETKOV, S. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proc. ACM SIGGRAPH* (Aug. 2001), p. to appear.

[114] RASO, M., AND FOURNIER, A. A Piecewise Polynomial Approach to Shading Using Spectral Distributions. In *Proc. Graphics Interface* (June 1991), pp. 40–46.

[115] REEVES, W., SALESIN, D., AND COOK, R. Rendering Antialiased Shadows with Depth Maps. In *Proc. SIGGRAPH* (July 1987), vol. 21, pp. 283–291.

[116] ROSSIGNAC, J., AND REQUICHA, A. Depth-Buffering Display Techniques for Constructive Solid Geometry. *IEEE CG&A 6*, 9 (1986), 29–39.

[117] RUSINKIEWICZ, S. A new change of variables for efficient BRDF representation. In *Eurographics Workshop on Rendering* (June 1998), pp. 11–23.

[118] SALISBURY, M., WONG, M., HUGHES, J., AND SALESIN, D. Orientable textures for image-based pen-and-ink illustration. In *Proc. SIGGRAPH* (Aug. 1997), pp. 401–406.

[119] SCHILLING, A., KNITTEL, G., AND STRASSER, W. Texram: A smart memory for texturing. *IEEE Computer Graphics and Applications 16*, 3 (May 1996), 32–41.

[120] SCHLICK, C. A customizable reflectance model for everyday rendering. In *Eurographics Workshop on Rendering* (June 1993), pp. 73–84.

[121] SCHRAMM, M., GONDEK, J., AND MEYER, G. Light Scattering Simulations using Complex Subsurface Models. In *Proc. Graphics Interface* (May 1997), pp. 56–67.

[122] SCHRÖDER, P., AND SWELDENS, W. Spherical Wavelets: Efficiently Representing Functions on the Sphere. In *Proc. SIGGRAPH* (Aug. 1995), pp. 161–172.

[123] SEGAL, M., AND AKELEY, K. *The OpenGL Graphics System: A Specification (Version 1.2.1)*, 1999.

[124] SEGAL, M., KOROBKIN, C., VAN WIDENFELT, R., FORAN, J., AND HAEBERLI, P. Fast shadow and lighting effects using texture mapping. In *Computer Graphics (SIGGRAPH '92 Proceedings)* (July 1992), pp. 249–252.

[125] SLOAN, P.-P. J., AND COHEN, M. F. Interactive Horizon Mapping. In *Rendering Techniques '00 (Proc. Eurographics Workshop on Rendering)* (2000), Springer, pp. 281–286.

[126] SLUSALLEK, P., STAMMINGER, M., HEIDRICH, W., POPP, J.-C., AND SEIDEL, H.-P. Composite Lighting Simulations with Lighting Networks. *IEEE CG&A 18*, 2 (Mar. 1998), 22–31.

[127] SNYDER, J. M. Interval analysis for computer graphics. In *Proc. ACM SIGGRAPH* (July 1992), vol. 26, pp. 121–130.

[128] SOLER, C., AND SILLION, F. X. Fast Calculation of Soft Shadow Textures Using Convolution. In *Proc. ACM SIGGRAPH* (1998), pp. 321–332.

[129] STÜRZLINGER, W., AND BASTOS, R. Interactive Rendering of Globally Illuminated Glossy Scenes. In *Eighth Eurographics Workshop on Rendering Workshop* (June 1997), Eurographics, pp. 93–102.

[130] TRENDALL, C., AND STEWART, A. J. General Calculations using Graphics Hardware, with Applications to Interactive Caustics. In *Rendering Techniques '00 (Proc. Eurographics Workshop on Rendering)* (2000), Springer, pp. 287–298.

[131] TUMBLIN, J., AND RUSHMEIER, H. Tone reproduction for realistic images. *IEEE CG&A 13*, 6 (Nov. 1993), 42–48.

[132] UPSTILL, S. *The RenderMan companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1990.

[133] VEACH, E., AND GUIBAS, L. Bidirectional Estimators for Light Transport. In *Fifth Eurographics Workshop on Rendering* (June 1994), pp. 147–162.

[134] VEACH, E., AND GUIBAS, L. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proc. SIGGRAPH* (Aug. 1995), pp. 419–428.

[135] VLACHOS, A., PETERS, J., BOYD, C., AND MITCHELL, J. L. Curved pn triangles. *2001 ACM Symposium on Interactive 3D Graphics* (March 2001), 159–166. ISBN 1-58113-292-1.

[136] VOORHIES, D., AND FORAN, J. Reflection Vector Shading Hardware. In *Proc. SIGGRAPH* (July 1994), pp. 163–166.

[137] WALTER, B., ALPPAY, G., LAFORTUNE, E., FERNANDEZ, S., AND GREENBERG, D. Fitting virtual lights For non-diffuse walkthroughs. In *Proc. SIGGRAPH* (Aug. 1997), pp. 45–48.

[138] WANGER, L. The effect of shadow quality on the perception of spatial relationships in computer generated imagery. In *SIGGRAPH Symp. on Interactive 3D Graphics* (Mar. 1992), pp. 39–42.

[139] WARD, G. Measuring and modeling anisotropic reflection. In *Proc. SIGGRAPH* (July 1992), pp. 265–272.

[140] WARD, G. Towards More Practical Reflectance Measurements and Models. In *Graphics Interface '92 Workshop on Local Illumination* (May 1992), pp. 15–21.

[141] WARD, G. The RADIANCE lighting simulation and rendering system. In *Proc. ACM SIGGRAPH* (July 1994), pp. 459–472.

[142] WESTIN, S., ARVO, J., AND TORRANCE, K. Predicting Reflectance Functions From Complex Surfaces. In *Proc. SIGGRAPH* (July 1992), pp. 255–264.

[143] WILLIAMS, L. Casting curved shadows on curved surfaces. In *Proc. SIGGRAPH* (Aug. 1978), vol. 12, pp. 270–274.

[144] WILLIAMS, L. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)* (July 1983), pp. 1–11.

[145] WOO, A., POULIN, P., AND FOURNIER, A. A Survey of Shadow Algorithms. *IEEE CG&A 10*, 6 (Nov. 1990), 13–32.